/N-60
/ 8574
69P

NASA Contractor Report 191461

TOOLS REFERENCE MANUAL FOR A REQUIREMENTS SPECIFICATION LANGUAGE (RSL), VERSION 2.0

G. L. Fisher
California Polytechnic State University
San Luis Obispo, CA

G. C. Cohen
Boeing Defense and Space Group
Seattle, WA

Contract NAS1-18586
November 1993

NASA
National Aeronautics and Space Administration

Langley Research Center
Hampton, Virginia 23681-0001

Table of Contents

1. Introduction

This report describes a general-purpose Requirements Specification Language, RSL. The purpose of RSL is to specify precisely the external structure of a mechanized system and to define requirements that the system must meet. A *system* can be comprised of a mixture of hardware, software, and human processing elements.

RSL is a hybrid of features found in several popular requirements specification languages, such as SADT (Structured Analysis and Design Technique [Ross 77]), PSL (Problem Statement Language [Teichroew 77]), and RMF (Requirements Modeling Framework [Greenspan 82]). While languages such as these have useful features for structuring a specification, they generally lack formality. To overcome the deficiencies of informal requirements languages, RSL has constructs for formal mathematical specification. These constructs are similar to those found in formal specification languages such as EHDM (Enhanced Hierarchical Development Methodology [Rushby 91]), Larch [Guttag 85], and OBJ3 [Goguen 88]. The formal features of RSL need not be used in all situations. A requirements specification can be constructed in a number of levels from informal, to semi-formal, to fully formal. Examples to follow will illustrate these levels.

The reader of this report is assumed to be familiar with concepts of computer programming, as well as concepts of discrete mathematics. No familiarity with the requirements analysis or formal specification is assumed. While the report is primarily a reference manual for the RSL language, it does include tutorial discussion and examples on the concepts of requirements specification.

1.1. The Overall Structure of a Requirements Specification

In RSL, a requirements specification consists of two parts: an *object-oriented* specification, and an *operation-oriented* specification. Dividing a specification into these two parts promotes the concept of multiple system views. One view focuses on the system from the perspective of the data objects, the other from the perspective of the functional operations. Neither view is *the* correct one -- they both convey the same meaning in different ways. Depending on the natural orientation of the system being specified, one form of view may be the more natural. Specifically, for so-called "transaction-oriented" systems, the object-oriented view is typically more natural. Conversely, for "transform-oriented" systems, the operation-oriented view is typically the more natural.

A transaction-oriented system is characterized by a large object on which relatively small, incremental operations are performed. A database management system (DBMS) is a good example of a transaction-oriented system. In a DBMS, a typically large database object is incrementally modified by relatively small operations to add, delete, and search for database entries. The DBMS operations are often referred to as transactions, and hence the system is referred to as transaction-oriented.

A transform-oriented system is characterized by relatively large operations that perform major transformations on objects. A report generation system (RPGS) is a good example of a transform-oriented system. In an RPGS, a large input object, such as a database, undergoes a major transformation to produce a large output object -- the report. As the name suggests, a transform-oriented system performs a major transformation on its input, such that the resulting output is substantially different in structure from the input. This is in contrast to a transaction-oriented system, in which the output is the result of a relatively small change to the input, and the input and output have much the same overall structure.

Whichever is the more natural view, an RSL requirements specification should always contain both. That is, if a system is primarily viewed as object-oriented, it will still contain an operation-oriented view. Conversely, a primarily operation-oriented specification will also contain an object-oriented view. In this way, the two views provide a form of cross checking on specification consistency and completeness. Something that may have been overlooked in the object-oriented view may arise naturally in the operation-oriented view, and vice versa. When complete, the pair of views provides the separate parts of a mutually consistent requirements specification.

1.2. Underlying Principles

RSL, and similar languages such as RMF and EHDM, share a number of common underlying principles. These principles include:

1. *Object/Operation Model* -- a specification is comprised fundamentally of two forms of entity: *objects* and *operations*; these entities have well-defined relations to one another.
2. *Hierarchy* -- the primary relation between entities of the same type is *hierarchy*; that is, an entity is composed hierarchically of subentities, which may in turn be further hierarchically decomposed.
3. *Input/Output Relationships* -- the primary relation between objects and operations is *input/output*; specifically, operations take objects as inputs and produce objects as outputs.
4. *Attribute/Value Pairs* -- in addition to relations to other entities, an entity may have other general *attributes* that further describe its properties and characteristics.
5. *Four Composition Primitives* -- when an entity is decomposed into subentities, four composition forms are used; the forms are:
    a. *and* composition: an entity is composed as a *heterogeneous collection* of subentities
    b. *or* composition: an entity is composed as a *selected one of* a heterogeneous collection of subentities
    c. *repetitive* composition: an entity is composed as a *homogeneous collection* of zero or more subentities
    d. *recursive* -- an entity may contain itself as a subentity.
6. *Class/Subclass/Instance Composition* -- a secondary form of hierarchical relation is that of *class/instance*; an entity *class* defines a generic entity template; an entity *subclass* or *instance* specializes the generic class by adding additional attributes.
7. *Object/Operation Duality* -- the composition and relational forms apply equally to both objects and operations; that is, both objects and operations can be decomposed hierarchically, with general attribute/value pairs, and defined as classes.
8. *Strong Typing* -- all objects in a specification define a formal *type*; RSL formulas and expressions are type checked to confirm that object names are used in type-correct contexts.

9. *Functional Operations* -- all operations are *fully functional* and *side effect free*; an operation may only access explicit input objects, and may effect change only through explicit output objects.

10. *Declarative Specification* -- a specification declares structure and function without specifying operational details.

1.3. Defining User-Level Specifications

It is possible to use RSL to specify a system on a number of levels. The most typical use of RSL is to define *user-level* specifications. Here the term *user* refers to the ultimate end-user of a system being specified. Hence, a user-level specification is one in which all objects and operations are directly accessible to an end-user.

The general guideline for developing user-level specifications is that if an end-user does not need to know about an object or operation, then that object or operation should not appear in a specification. The purpose of this guideline is to keep an RSL specification as free as possible of implementation detail. As a *specification language*, RSL should be used to define *what a system does*, not *how the system works*.

For most systems, the dividing line between what an end-user does and does not need know is reasonably clear. As an example, consider the specification of two different mechanized systems: an automobile and a database management system. Table 1 summarizes what objects and operations end users should know about in these two systems.

The table illustrates the spectrum of information an end-user needs to know. It is clear that an end-user absolutely needs to know about the basic objects and operations of the mechanized system. These basics are the objects and operations specifically intended to be the interface between the end-user and the system itself. In the case of an automobile system, the basics include the doors, steering wheel, keys, etc. Basic user-level operations include starting the car, driving it, and stopping. In a computerized DBMS there are comparable user-level objects and operations. The objects include the database records, and the record fields such as name, address, etc. DBMS user-level operations include adding, deleting, and searching for a record.

In addition to basic objects and operations, it is frequently necessary for end users to be aware of certain system properties. The table shows examples of such user-level properties. For the automobile, the user should probably be aware that the automobile uses unleaded fuel. For the DBMS, the user should probably be aware that duplicate records cannot be entered into the database. It is conceivable that a very naive or uninformed user could avoid knowing such properties of a system. Hence, the table indicates the user "probably" should know. At any rate, the system specification should include these as user-level information, whether or not a user chooses to know the information.

The next entries in the table exemplify information that the end-user probably need not know. The example in this category for the automobile is the fact that the automobile uses spark plugs. For the DBMS, the example is that the database uses an ISAM format (Indexed Sequential Access Method). Such information is more appropriately directed to the system maintainers rather than end users. It is implementation-level in nature.

It might be argued that end-users should know information in the "probably not" category on the grounds of performance. That is, an end-user may better understand how the system will perform if certain internal details of the system are made known. However, it is more appropriate that performance information be stated in terms that leave out implementation detail. For example, specifying that the DBMS uses ISAM may indirectly help a knowledgeable end-user

Automobile Spec	DBMS Spec	Does User Need to Know About It?
Accessible objects: doors, steering wheel, keys. Accessible operations: start car, drive car, stop car.	Accessible objects: data records, name field, address field. Accessible operations: add record, delete record, find record.	Absolutely.
The automobile operates on unleaded gasoline.	Duplicate record entries are not allowed in the database.	Probably.
The engine has spark plugs.	Database entries are in ISAM format.	Probably Not.
A spark plug fires at 5000V.	An ISAM record takes 32 sectors.	Absolutely Not.

Table 1: Levels of what Users May Need to Know.

know which forms of query operation will be efficient and which inefficient. However, many end-users may not understand at all what ISAM formatting means. It would be better to omit the ISAM detail from the user-level specification altogether. The specification should rather define operation performance properties directly in terms that all end-users can understand. For example, searching for a record by key 1 will take less time than searching for the same record by key 2.

The last entry in the comparison table above indicates information that clearly need not be known by an end user. Such information relates to low-level system implementation details that end users need never be aware of.

1.4. Relationship between an RSL Specification and a Concrete User Interface

Relating an RSL specification to a concrete user interface is another means to determine what is appropriate user-level information. Specifically, any user-level object or operation should be directly visible in the user interface. Objects appear in the interface as any visible data values. These include data that are displayed on a screen, in a hardcopy form, or stored in some user-accessible data file. Operations appear in the interface as any form of user command. In concise terms, an operation is invoked as the result of some *gesture* that is performed by the user. Gestures include keyboard typing, selecting from a mouse-oriented command menu, or pressing a function button with a mouse or other pointing device.

RSL has also been applied to a large-scale requirements specification for an advanced subsonic civil transport system. This specification appears in Appendix A of [Frincke 92].

1.7. The RSL Toolset

RSL is supported by a set of tools that perform specification checking and provide browsing capabilities. The three major tools are:

1. The Basic RSL Translator
2. The RSL Text Browser
3. The RSL Graphic Browser

The Basic Translator provides facilities similar to a programming language compiler -- syntax analysis, type checking, and interactive interpretation. The browsers provide two different views of an RSL document. The Text Browser allows users to navigate through an RSL specification using a number of menus and textual links. The Graphic Browser allows users to view supporting pictures that aid in the understanding of the text.

The tools are described in full detail in a companion reference manual [Fisher 93].

2. Syntactic and Lexical Elements of RSL

Like a programming language, RSL has a formal syntax. The complete RSL syntax is given in Appendix D. The syntactic notation is a style of BNF, as is used to define programming language syntax. Readers unfamiliar with BNF notation should consult a text on programming languages, or other suitable reference.

The lexical elements of RSL are similar to the lexical elements of a programming language. RSL specifications contain identifiers that denote the objects, operations, and other named components of the specification. Literal values, such as numbers and strings, are used in an RSL specification. The formal sections of a specification contain expressions that use operator symbols similar to those available in a programming language. These and other lexical elements are defined in the following subsections.

2.1. Identifiers

An RSL identifier starts with a letter, followed by zero or more letters, digits or underscore characters, followed at the end by an optional single-quote character. Letters include both upper and lower case letters, "A" through "Z" and "a" through "z". Digits are "0" through "9". The underscore character is "_". The single quote character is "'".

The intent of the trailing single quote in an identifier is to provide a *prime* notation. For example, if "Name" is an identifier in a specification, then "Name'" is read *Name prime*. "Name" and "Name'" will typically be used in a related context. For example if "Name" is the input to an operation, then "Name'" could be an output. Later examples illustrate typical uses of the prime notation.

Examples of legal identifiers are the following:

```
Name    Name'    xyz    XYZ    Object25    operation_15
```

2.2. Literal Values

Literal values in RSL are numbers, strings, booleans, and empty values. Numbers are composed of one or more digits, with an optional embedded decimal point. Numbers must begin and end with a digit. Hence, 1.0 and 0.1 are legal numbers, but 1. and .1 are not legal.

All objects and operations should be traceable to a concrete user interface. However, this does not mean that an RSL specification should define concrete interface details. In fact, RSL specifications should be *free of interface details*. It is intended that RSL be used to define *abstract* user-level specifications, for which any number of concrete interfaces may be defined.

Consider the DBMS example above. An RSL specification will define the abstract structure of the database and record objects, as well as the abstract input/output behavior of the database operations. Given this abstract specification, a number of concrete interfaces could be defined. For example, a text-command interface would define concrete names for the operations, such as "add", "del ", "search". A typical add command could be typed in as follows:

```
add Name=Smith, Age=25 to DB5
```

This concrete command corresponds to the invocation of an abstract AddRecord operation, where AddRecord is the operation name defined in the RSL specification.

A quite different form of window-based interface could also be defined for the same abstract DBMS specification. In such an interface, the "add", "del", and "search" commands could be pull-down menu selections. The equivalent of the above text command would involve a pull-down gesture sequence to select "add" from the command menu. This would be followed by information entry in some dialog window that would appear in response to the command menu selection. While selecting "add" from a pull-down menu is concretely different than typing "add ..." as a text command, both concrete interface commands correspond to exactly the same abstract operation, namely AddRecord.

An indication that a specification contains inappropriate interface detail is the appearance of objects with names such as "CommandLineText" or "PullDownMenuItems". Such objects should not be in a specification if they refer to the details of a particular style of concrete interface.

In summary, an RSL specification should be an abstract, user-oriented definition of a system. The specification should be free of implementation detail in any form. The specification should also be free of concrete user interface details.

1.5. Defining Specifications for Embedded Systems

A so-called embedded computer system is intended to exist within some other engineered environment. Embedded systems are frequently characterized by little or no interaction with a human end user. The guidance system for a autonomous vehicle is a good example. In such systems, the preceding guidelines for user-oriented specification must be considered in a different light.

For embedded systems, the "users" are those external components of the surrounding environment with which the embedded system must interface. Identification of "external" may be somewhat more subjective in an embedded system, without the specific focus of a human user. However, the general guideline for embedded system specification is the same as discussed above. Namely, the requirements specification should define the externally visible objects and operations, for some appropriate definition of externality.

1.6. Suitable Application Domains

RSL is intended to be a general purpose language. As such, it is suitable for specifying the software and/or hardware components of any computer-based system. In practice, RSL has been used extensively in courses on Software Engineering to specify medium-scale software systems.

String-valued literals are surrounded with double quote characters. A double quote character can be embedded within a string by preceding it with a backslash character.

A boolean literal is denoted by the standard identifier **true** or **false**. The empty value is denoted by the standard identifier **nil** or **empty**.

Examples of legal literal values are the following:

```
1    1234    1.0    123.4567    "abc"    ""    "Say \"hey\""
true    false    nil    empty
```

2.3. Punctuation Symbols and Expression Operators

Table 2 summarizes the use of the RSL syntactic punctuation symbols. Further examples of the use of punctuation symbols appear throughout the report.

RSL definitions contain symbolic expressions in a number of contexts. These expressions are similar to expressions in programming languages. The legal operator symbols in RSL are the following:

```
+   *   -   /   (   )   =   ~=   <>   <   >   <=   >=   .   ?
[   ]   ==   and   &   or   |   not   =>   implies   <=>   iff
forall   exists
```

Later sections of the report define the meanings of these symbols and provide examples of their use in expressions.

The reader should not confuse the RSL terms "operation" and "operator". An operation is a user-defined component of an RSL specification. An operator is a symbol used within an expression.

As in many programming languages, some symbols in RSL are *overloaded*. That is, the same symbol has two different meanings depending on the context of its use. For example, the comma is an overloaded symbol in RSL. In some contexts it is used as a separator and in other

Symbol	Usage
;	A semicolon separates major definitional components. For example, each attribute of an object and operation definition is separated by a ";". Also, object and operation definitions themselves are separated by a ";".
:	A colon separates pairs of syntactic items. For example, attribute/value pairs are separated by a ":".
,	A comma separates items in a list of syntactic elements. For example, the list of operations associated with an object is comma-separated.

Table 2: Summary of Punctuation Symbols.

contexts as an expression operator. Examples in later sections of the report illustrate the use of all operators. The formal syntax in Appendix D defines all syntactically legal usages.

2.4. Comments

RSL comments are enclosed in two forms of bracket pairs: (* ... *) or { ... }. There is no difference between the two forms of comments; there are two different forms for historical reasons. Comments may not be nested.

2.5. RSL Keywords and Standard Identifiers

RSL keywords are distinguished words that can only be used in particular syntactic contexts. Keywords cannot be used as identifiers. The following are the legal RSL keywords:

```
and   attribute   ax   axiom   class   collection   components
define   else  equations   exist   exists   forall   func
function   if   iff   implies   in   inputs   outputs   instance
is   end   list   module   nil   not   obj   object   of   op
operation   operations   ops   or   out   parts   post postcond
postcondition   postconditions   pre   precond   precondition
preconditions   then   there   var   variable
```

RSL standard identifiers are names that have pre-defined meanings. The difference between keywords and standard identifiers is that keywords appear in particular syntactic contexts, whereas standard identifiers appear in the more general syntactic context of an identifier. Other than syntactic context, there is no practical difference between keywords and standard identifiers, since neither class of symbol can be redefined by the user. The RSL standard identifiers are the following:

```
boolean   empty   false   integer   nil   none   real   string
true   description   descrip   picture   pic
```

3. Objects and Operations

As noted in the introduction, the primary components of an RSL definition are objects and operations. Objects and operations have similar forms of definition, as the following subsections illustrate.

3.1. Defining Objects

An object is specified in a fixed format showing its components and other attributes. The general form is as follows[1]:

object *name* **is**
  components: *composition expression defining subobjects*;
  operations: *list of applicable operations*;
  equations: *formal equations for operations*;
  [**description:** *free-form text*;]
  [*other attributes: user-defined information*;]
end *name*

The **components** attribute of an object defines the subobjects of which it is composed. The

[1] boldface terms are keywords, italic terms are variables, optional terms are enclosed in square brackets [...]

operations attribute lists all operations that construct or access the object. The equations attribute defines formal algebraic equations that specify the precise meaning of the object in terms of its operations. The **description** attribute is a prose description intended to convey the structure and meaning of the object to a human reader. *Other attributes* can be defined by the user to help specify an object more clearly.

The following are example object definitions:

```
object PersonDatabase is
    components: collection of PersonRecord;
    operations: AddRecord, DeleteRecord, FindRecord, CreateDatabase;

    description: {
        A PersonDatabase contains zero or more personal information
        records.
    };
end PersonDatabase;

object PersonRecord is
    components: Name and Age and Address;
    description: {
        A PersonRecord contains the information items for an
        individual in the PersonDatabase.
    };
end PersonRecord;

object Name is string;
object Age is number;
object Address is string;
```

These examples show the basic structure of an object in terms of its components, operations, and description. Equation definitions are an advanced topic, covered in Section 7 of the report. The definition of other attributes is covered in Section 6.

Each object definition in a specification defines an individual *type* of object. In relationship to a typed programming language, object definitions in RSL are analogous to type definitions in a programming language. That is, each object defines a type structure for a typically infinite set of concrete values. RSL types are more abstract than programming language types, as will be discussed later.

3.2. Defining Operations

An operation specification is much like an object specification, in a fixed format of components and other attributes. Here is the general form:

operation *name* **is**
  components: *composition expression defining suboperations* ;
  inputs: *list of objects*;]
  outputs: *list of objects*;]
  preconditions: *formal predicate on inputs*;]
  postconditions: *formal predicate on inputs and outputs*;]
  [**description:** *free-form text*;]
  [*other attributes: user-defined information*;]
end *name*

For example, here are some companion operations to the earlier object examples:

```
operation AddRecord is
    inputs: PersonDatabase, PersonRecord;
    outputs: PersonDatabase;
    description: {
        Add a new record into a person database.
    };
end AddRecord;

operation DeleteRecord is
    inputs: PersonDatabase, Name;
    outputs: PersonDatabase;
    description: {
        Delete an existing record by Name.
    };
end DeleteRecord;

operation FindRecord is
    inputs: PersonDatabase, Name, PersonRecord;
    outputs: PersonRecord;
    description: {
        Find an existing record by Name.
    };
end FindRecord;

operation CreateDatabase is
    inputs: none;
    outputs: PersonDatabase;
    description: {
      Create an initially empty person database.
    };
end CreateDatabase;
```

3.3. Component Expressions

The components of an object or operation are defined using a *composition expression*. There are four forms of composition, as outlined in the introduction of the report: *and*, *or*, *repetitive*, and *recursive*. Table 3 summarizes the RSL symbols used in composition expressions. The symbols for the same composition form are synonyms. That is, ',' and the keyword **and** have exactly the same meaning. The synonymous forms exist simply for stylistic convenience.

The names that appear in a composition expression are the names of other defined entities. Consider the following example:

```
object O is
    components: O1 and O2 or O3;
end O;
```

This definition defines an object named O, containing three other objects as its components. Objects O1, O2, and O3 must in turn be defined in order for the specification to be complete. The chain of subobject definitions ends with the definition of *atomic* objects, as will be discussed shortly.

The precedence of composition operators from lowest to highest is: or, and, '*'. Composition expressions can include parentheses, in the normal way, for grouping and changing precedence. For example,

```
(A or B) and (C or D).
```

Symbols	Composition Form	Examples
',', and	*and* composition	A and B and C A, B, C
'|', or	*or* composition	A or B or C A | B | C
list of, '*', collection of	*repetitive* composition	A* list of A collection of A

Table 3: Summary of Component Expression Symbols.

is a legal composition expression.

While any level of parenthesis nesting is possible, it is recommended that parentheses be used sparingly in practice. For example, consider the following definition:

```
object Database is
    components: (Name and Addr and Age)*;
end DB;
```

A better alternative is:

```
object Database is
    components: Record*;
end DB;

object Record is
    components: Name and Addr and Age;
end Record;
```

The latter alternative is clearer, and promotes reuse of the object named Record.

Each of the four composition primitives defines a particular structure. Used in combination, the composition primitives can define a wide variety of structures commonly used in specifications. The following subsections provide further details and examples on the use of the RSL composition forms.

3.3.1. And Composition

The and composition operator defines a heterogeneous collection of objects. In mathematical terms, an and-structure is a tuple. In relation to a programming language, an and-structure is analogous to a record type. That is, the and operator defines an object with a fixed number of components, where the components may be any other type of object. For example, the following definition specifies an object with exactly three components:

```
object A is
    components: X and Y and Z;
end A;
```

The components of an and-structure are never optional. That is, each component is always present.

3.3.2. Or Composition

The or composition operator also defines a heterogeneous collection of objects. In mathematical terms, an or-structure is a tuple of boolean-tagged elements, where exactly one of the elements is tagged true, and all other pairs are tagged false. The true tag indicates which of the or-structure elements is present. In relation to a programming language, an or-structure is analogous to a union type or variant record.

The following definition specifies an object with one of three possible components:

```
object O is
    components: X or Y or Z;
end O;
```

In contrast to an and-structure, only one of the components of an or-structure is present at one time.

3.3.3. Repetitive Composition

The repetitive composition operator defines a homogeneous collection of zero or more objects. In mathematical terms, a repetitive structure is an ordered bag (a bag is a set with duplicate entries allowed). In relation to a programming language, a repetitive structure is analogous to a list or array. However, a repetitive structure differs significantly from an array in that a repetitive structure is not of a fixed size.

The following definition specifies an object with zero or more components:

```
object L is
    components: X*;
end L;
```

3.3.4. Recursive Composition

Unlike the preceding three composition forms, recursive composition has no explicit composition operator. A recursive definition results when an object is defined such that it contains itself as a component, or subcomponent. In mathematical terms, a recursive structure corresponds to a recursive mathematical definition. In relation to a programming language, a recursive structure corresponds to a recursive type definition, which is typically defined using a pointer. However, a recursive structure in RSL is *not* a pointer-based definition. RSL contains no pointers.

The following are examples of recursive definitions:

```
object R is
    components: X and Y and R;
end R;

object R1
    components: R2 and R3 and R4;
end R1;

object R2 is
    components: R5 and R6;
end R2;
```

```
object R5 is
    components: R7 and R1;
end R5;
```

In object R, the recursion is direct, since R is defined immediately as a component of itself. The object R1 is indirectly recursive -- R1 has a component R2, which in turn has a component R5, which in turn has a component R1.

3.3.5. Common Uses of RSL Composition Forms

The PersonDatabase object in Section 3.1 is a good example of and-composition. An example of or-composition is

```
object MaritalStatus is
    components: Married or Unmarried or Widowed or Divorced;
end MaritalStatus;
```

It is common in specifications to define an object that contains any of a number of subobjects, intermixed in any order. Consider the following example:

```
object UnformatedDocument is
    components: (RawText or FormattingCommand)*;
    description: (*
        An UnformatedDocument contains RawText strings and
        formatting commands intermixed in any order.
    *);
end UnformatedDocument;
```

The general composition form that specifies intermixed components is:

```
components: (A or B or ... )*;
```

This expression specifies a component structure that has an A or B or ..., any number of times, in any order. It should be noted that the following form does not specify intermixing:

```
components: (A* and B* and ...)
```

This form specifies a component structure that has zero or more A's, followed by zero or more B's, followed by ... . That is, all of the A's, if any, come first, followed by all of the B's, etc. Keep in mind that and-composition specifies a tuple, where each component of the tuple must be present.

The following example illustrates a practical use of recursive composition:

```
object DatabaseQuery is
   components: SimpleQuery |
                  SimpleQuery, OrQueryOperator, DatabaseQuery
   description: (*
        A DatabaseQuery is an object that specifies how a database
        search can be conducted.  For example, if a user wanted to
        find all the records in a database with a name of Smith and
        age of 25, or a name of Jones and age 35, the DatabaseQuery
        object would be
               Name=Smith and Age=25 or Name=Jones and Age=35
    *);
end DatabaseQuery;

object SimpleQuery is
    components: FieldValueSpecifier |
                 FieldValueSpecifier, AndQueryOperator, SimpleQuery
    description: (*
```

```
        A SimpleQuery is the component of a query that and's
        together two or more FieldValueSpecifiers, or is just a
        single FieldValueSpecifier.
    *)
end SimpleQuery;

object FieldValueSpecifier is
    components: FieldKey, EqualsSign, FieldValue;
end FieldValueSpecifier;

object FieldKey is
    components: NameKey | AgeKey | AddressKey;
end FieldKey;

object NameKey = "name";
object AgeKey = "age";
object AddressKey = "address";
object EqualsSign = "=";
object OrQueryOperator = "or";
object AndQueryOperator = "and";
```

3.3.6. Composition Expressions in Operations

In the current version of RSL, fully general composition expressions are only meaningful in object definitions. In operation definitions, only **and**-composition is meaningful. The use of **or**-composition in an operation definition is equivalent to **and**-composition and the use of **list**-composition is ignored. For example, in RSL Version 2, the following two operation definitions are equivalent:

```
operation O1 is
    components O2 or O3 or O4*;
end O1

operation O1 is
    components O2 and O3 and O4;
end O1
```

In a future version of RSL, general composition of operation components may be supported.

3.4. Operation Typing and Functionality

As in a programming language, operation input/output lists define the formal parameter types that are accepted and produced by an operation. In contexts where operations are used with actual parameters, the same sort of type checking applies in RSL as in a programming language. Namely, the type of each actual parameter must agree with the type of the corresponding formal parameter.

As noted in the introduction, all the operations defined in a specification are *side effect free*. This means that an operation cannot use any object unless that object is an explicit input. Further, an operation may effect change only through an explicit output. In the nomenclature of programming languages, RSL is a *fully functional* language.

The notion that operations *effect change* rather than modify objects is also an important aspect of functional definition. An operation does not modify objects to produce output objects. Rather, a fully functional operation can only *create new objects*.

Consider the AddRecord example defined above. When this operation executes, it accepts a database and record as inputs. What it outputs is a *new copy* of the input database, with a *new* *copy* of the input record added into the database. In programming language terms, functional specifications have no global variables, no global files, and no call-by-var parameters. In this sense, RSL functional definitions are similar to definitions in functional programming languages such as pure LISP and ML.

The fully functional specification of operations is sometimes counter-intuitive, particularly in the case of large objects in a transaction-oriented system. For example, one might consider the explicit input and output of a large database to be unnecessary and/or inefficient. It is necessary since in order to construct a result that contains a new record, the original database must be input. It cannot be assumed that the operation will read from some stored database file or other external storage structure.

With regards to implementation efficiency, this matter is strictly not of concern in an RSL specification. It is almost certainly not the case that a DBMS implementation would copy entire databases from input to output. However, such implementation concerns are beyond the scope of an RSL specification. The specification states in functional terms what an operation does, including all inputs and outputs that it uses. A subsequent implementation can use whatever efficient techniques are available, as long as the implementation meets the abstract specification.

3.5. Names and Types

In the examples thus far, the components of an entity have been shown as simple names. Consider the PersonRecord example from above:

```
object PersonRecord is
    components: Name and Age and Address;
    description: (*
        A PersonRecord contains the information items for an
        individual in the PersonDatabase.
    *);
end PersonRecord;
```

Here Name, Age, and Address are the names of other defined objects. Consider the following alternate definition of PersonRecord:

```
object PersonRecord is
    components: n:Name and a:Age and ad:Address;
    description: (*
        A PersonRecord contains the information items for an
        individual in the PersonDatabase.
    *);
end PersonRecord;
```

Here the components are defined using *name/type* pairs. The component structure of PersonRecord is precisely the same in both of the above two definitions. The *name* component of the name/type pair is a local subobject name by which the component can be referenced. The names are *n*, *a*, and *ad* in this example. The *type* half of a name/type pair is the name of a defined object. The object types in this example are *Name*, *Age*, and *Address*.

A components definition can be legally specified with or without name/type pairs. Name/type pairs are used when it is necessary to refer to a component in an RSL expression. Name/type pairs can also be used in class definitions, and other RSL contexts. Upcoming sections of the report discuss the uses of name/type pairs in further detail.

It is instructive to contrast the use of name/type pairs in RSL versus a programming language. Consider, for example, the equivalent of the last PersonRecord definition in a Pascal-like language:

```
type Name = string;
type Age = integer;
type Address = string;
type PersonRecord =
    record
        n: Name;
        a: Age;
        ad: Address;
    end;
```

Each RSL object is defined as a type in the programming language. Except for notational differences, the RSL and programming language definition have the same meaning.

A difference between RSL and a Pascal-like language is that component names are not required in RSL. Consider the following fictitious Pascal-like record definition, which is equivalent to the original RSL definition of PersonRecord, and which uses the equivalent of RSL opaque atomic types (defined in Section 3.6 below):

```
type Name;
type Age;
type Address;
type PersonRecord =
    record
        Name;
        Age;
        Address;
    end;
```

In this example, the field names are omitted in the record definition. This is generally not allowed in a programming language, since it precludes runtime access to the record fields. However, such nameless fields are perfectly reasonable in RSL, since there is no "runtime" to worry about. That is, the structure of an RSL object need only declare what the components are, without necessarily providing a means to access the components. In this sense, an RSL definition can be more abstract than a corresponding programming language definition.

There are cases in which object component access *is* necessary in RSL, in which cases component names *are* necessary. These cases will be discussed in detail in upcoming examples.

3.6. Composite versus Atomic Objects

Any object defined with a non-empty components field is *composite*. An *atomic* object is one of the following:

- an object defined as a built-in type,
- an object with no components field, or
- an object defined with "**components: empty**".

The built-in atomic types are **number**, **integer**, **real**, **string**, and **boolean**. The **number** atomic type is a synonym for **real** -- they both represent mathematical real numbers. The **integer** type is the normal subset of **real**. The **string** type represents symbolic values. The **boolean** type denotes a true/false value.

The general format for a composite object definition is the following:

object *name* **is**
  components: ...;
    . . .

end *name*

In contrast, the general format for an atomic object definition is:

object *name* **is** *name*
  [. . .
end *name*]

where the definition body (denoted by the ellipses) has no **components**. The *name* following the **is** must be a built-in atomic object name or the name of another atomic object. Note that the body in an atomic object definition is optional. Hence, an atomic definition may take a simple form, such as:

```
object DataMaximum is number;
```

or a longer form, such as:

```
object DataMaximum is number
    operations: . . . ;
    description: . . . ;
end DataMaximum;
```

The reader should note the distinction between the following two definitions, the first of which is composite and the second atomic:

```
object C is
    components: integer;
end C

object A is integer;
```

`C` is a composite object containing a single integer component. In contrast, object `A` is an atomic integer object. While the definitional form of object `C` is not particularly useful, it is syntactically valid and must therefore be understood.

A special form of atomic definition is an *opaque* object. The general form of opaque definition is either of the following:

object *name* ;

object *name* **is**
  components: ;
  . . .
end *name*

As in the preceding form of atomic definition, the body in an opaque definition is optional, so that any one of the following forms is legal:

```
object ExternalItem;

object ExternalItem is
    description: ... ;
end ExternalItem;

object ExternalItem is
    components: ;
    description: ... ;
end ExternalItem;
```

If the **components** keyword is given in an opaque definition, it must be followed by an empty components expression, or by one of the single keywords **empty** or **none**. The keywords **empty** and **none** are synonyms, both denoting an empty composition expression. The use of opaque definitions is discussed further in Section 3.8.

3.7. Abstract versus Concrete Objects

Object definitions that use the keyword **is** define a formal object *type*. Object definitions that use the symbol "=" in place of **is** define a concrete object *value*. A type represents an abstract set of many possible values, typically an infinite set. A value represents a single concrete object. For example, the standard type **integer** represents the infinite set of all possible integers. The value 10 represents the single integer value 10.

As a user-defined example, the following object definition represents the set of all possible values that have a number component and a string component:

```
object NumberAndString is
    components: number and string;
end NumberAndString;
```

In contrast, the following definition defines a single value of this type:

```
object TenAndXyz:NumberAndString = [10, "xyz"];
```

There are two general formats for concrete value definition:

> **object** *value-object-name* = *value-expression*
>   [. . .
> **end** *name*]

and

> **object** *value-object-name*:*type-object-name* = *value-expression*
>   [. . .
> **end** *name*]

The first of these two forms is used to define an object as a value of an atomic type, i.e., numeric, string, or boolean. The second form is used to define concrete values of composite types. In both forms, additional object attributes are optional.

The *value expression* to the right of the "=" denotes the concrete value. A numeric value is denoted by a real or integer numeric literal. A string is denoted by a double-quoted string literal. A boolean value is denoted by one of the standard identifiers **true** or **false**. A concrete composite value is constructed using square bracket operators "[" and "]". For example, the object value [1, 2, 3] is a composite value consisting of the numbers 1, 2 and 3. Composite values can be nested to any depth.

A composite value can represent either an *and*'d or *list-of* object as necessary. For example, the value [1, 2, 3] is a concrete value for either of the following types of object:

```
object NumberList is
    components: number*;
end NumberList;

object NumberTriple is
    components: number and number and number;
end NumberTriple;
```

3.8. The Representational Base of Object Hierarchies

In order for an RSL definition to be complete, all referenced entities must be defined. In particular, there must be a definition for any object referenced by name in the components part of another definition or in the input/output lists of an operation. At some point, the hierarchy of object definitions must "bottom out" at atomic objects. Given the forms of atomic object discussed above, there are three levels of abstraction for defining the atomic basis of an object hierarchy:

1. *opaque* objects
2. atomic objects defined as one of the built-in atomic types
3. atomic objects defined as a specific concrete value

Of the three alternative levels, opaque definitions are the most abstract. An opaque definition signifies that the value set of an object is to be considered implementation-dependent, and that the abstract specification will not specify it concretely. Defining an object as a built-in type is intermediate in abstraction. This form of definition is more concrete than an opaque definition, but less concrete than specifying an object as a specific concrete value. Defining as concrete values is the most concrete basis for a requirements specification.

As an example, recall the DatabaseQuery definition from above. In that example, the atomic objects NameKey, AgeKey, etc. were defined as concrete atomic values. They could alternatively have been defined as opaque objects using the following definitions, in place of the definitions given originally in the example:

```
object NameKey;
object AgeKey;
object AddressKey;
object EqualsSign;
object OrQueryOperator;
object AndQueryOperator;
```

The third alternative for these atomic objects is the following set of definitions:

```
object NameKey is string;
object AgeKey is string;
object AddressKey is string;
object EqualsSign is string;
object OrQueryOperator is string;
object AndQueryOperator is string;
```

Any of the three levels of atomic specification is legal in RSL. Which is chosen depends on the degree of abstraction desired.

3.9. Default Operation Parameters

When concrete objects are defined in a specification, it can be useful to specify operations that accept these objects as inputs and/or produce them as outputs. Consider the following addition to the PersonDatabase example presented earlier:

```
object DefaultName = "Name Unknown";
object DefaultAge = -1;
object DefaultAddress = "Address Unknown";
object DefaultPersonRecord = [DefaultName, DefaultAge, DefaultAddress];

operation AddRecord is
    inputs: PersonDatabase, PersonRecord:=DefaultPersonRecord;
    outputs: PersonDatabase;
    description: (*
        Add a new record into a person database.
    *);
end AddRecord;
```

Here, concrete objects have been defined as defaults for the components of a default PersonRecord. The AddRecord operation has then been modified to specify a default value for the PersonRecord input.

The general format for defining default operation parameters is the following:

operation *name* **is**
  inputs: *type-name:=value-name, ...* ;
  outputs: *type-name:=value-name, ...* ;
end *name*;

where *type-name* denotes an abstract object and *value-name* denotes a concrete object.

A default value in a parameter list specifies what the standard parameter value should be, if no other value is given when an operation is invoked. Using parameter defaults, a specifier can define specific concrete values that constitute the standard required data for a system. When the system is implemented, these standard data will be those installed for initial system operation. In the absence of other user input that changes these values, the defaults will remain in use.

4. Classes

An object or operation definition may be specified as a *class*. A class definition is a general template for an object or operation, of which more specific *instances* can be declared. The following example shows a redefinition of the PersonRecord object defined earlier, this time as a class:

```
object class PersonRecord is
    components: Name and Age and Address;
    description: (*
        The PersonRecord class contains components that are common
        to all personnel in the database.
    *);
end PersonRecord;
```

This example specifies that all records in the database will contain the common components of Name, Age, and Address. The PersonRecord class can then be *specialized* as follows:

```
object StaffEmployee instance of PersonRecord is
    components: HourlyWage and EmploymentStatus;
    operations: ;
    description: (*
        A StaffEmployee is distinguished by HourlyWage and
        EmploymentStatus components.
    *);
end StaffEmployee;

object Programmer is
    components: Salary and Step;
    operations: ;
    description: (*
        A Programmer is distinguished by Salary and Step
        components.
    *);
end Programmer;

object Manager instance of PersonRecord is
    components: Salary and Step and Supervisees;
    description: (*
```

```
      A Manager is distinguished by Salary, Step, and Supervisees
           components.
     *);
  end Manager;

  object Supervisees is
      components: (StaffEmployee or Programmer)*;  --
      description: (*
        This is the list of people that a manager supervises.
      *);
  end Supervisees;
```

Each instance of PersonRecord is said to *inherit* the generic class components. That is, StaffEmployee, Programmer, and Manager all inherit the Name, Age, and Address components from the PersonRecord class. In addition, each of the instances further specializes itself by adding further fields. For example, a StaffEmployee is specialized by HourlyWage and EmploymentStatus components.

The purpose of a class is to define attributes that are common to a number of entities. The class contains the common attributes, and each instance automatically inherits these attributes, in addition to adding zero or more specializing attributes.

Classes may be defined in any number of levels. Consider the following refinement of the preceding example:

```
  object class PersonRecord is
      components: Name and Age and Address;
      description: (*
          The PersonRecord class contains components that are common
          to all personnel in the database.
      *);
  end PersonRecord;

  object StaffEmployee instance of PersonRecord is
      components: HourlyWage and EmploymentStatus;
      operations: ;
      description: (*
          A StaffEmployee is distinguished by HourlyWage and
          EmploymentStatus components.
      *);
  end StaffEmployee;

  object class SalariedEmployee instance of PersonRecord is
      components: Salary and Step;
  end SalariedEmployee;

  object Programmer instance of SalariedEmployee is
      description: (*
          A Programmer is now just an instance of SalariedEmployee,
          from which in inherits components Salary and Step components.
      *);
  end Programmer;

  object Manager instance of SalariedEmployee is
      components: Supervisees;
      description: (*
        A Manager inherits Salary and Step components.  It
          specializes with Supervisees.
```

```
    *);
end Manager;

object Supervisees is
    components: (StaffEmployee or SalariedEmployee)*;
    description: (*
      This is the list of people that a manager supervises.
    *);
end Supervisees;
```

Here there are three levels of class definition. A class object that is itself an instance is called a *subclass*. In this case, SalariedEmployee is a subclass. It inherits components from PersonRecord and in turn defines components that will be shared by its instances. Note that inheritance is fully transitive. That is, instances inherit all components from all levels of parent class above them. In this example, the instance object Programmer inherits all components from the two levels of parent class above it. Namely, it inherits Name, Age, Address, Salary, and Step. Note further that an instance need not provide any specializing components if it inherits all that it needs from its parent class(es). The Programmer object is such a case.

Using name/type pairs in a class definition can enhance the expressibility of the class hierarchy. Suppose it is the case that all employees have an office, but the specific type of office is defined differently for individual instances. The following example expresses this idea:

```
object class PersonRecord is
    components: Name and Age and Address and Office: ;
    description: (*
        The PersonRecord class contains components that are common
        to all personnel in the database.
    *);
end PersonRecord;

object StaffEmployee instance of PersonRecord is
    components: HourlyWage and EmploymentStatus and Office:none;
    operations: ;
    description: (*
        A StaffEmployee is distinguished by HourlyWage and
        EmploymentStatus components.
    *);
end StaffEmployee;

object class SalariedEmployee instance of PersonRecord is
    components: Salary and Step;
end SalariedEmployee;

object Programmer instance of SalariedEmployee is
    components: Office: SharedOffice;
    description: (*
        A Programmer is now just an instance of SalariedEmployee,
        from which in inherits Salary and Step components.
    *);
end Programmer;

object Manager instance of SalariedEmployee is
    components: Supervisees;
    description: (*
```

```
    A Manager inherits Supervisees
        components.
  *);
end Manager;

object SharedOffice is
    components: OfficeNumber and Desk*;
end SharedOffice;

object PrivateOffice is
    components: OfficeNumber and Desk and Window;
end PrivateOffice;
```

Here the "Office: " component of PersonRecord is only the name half of a name/type pair. Syntactically, it is a name followed by a colon, with no object name following the colon. This means that an instance must specify the type half of this pair to complete the definition. This form of component in a class definition is called a *specialization-required* component.

In the above example, the StaffEmployee object is specialized with "Office:none". Note that since specialization is required for the Office, a complete definition must define the type for Office:, even if that type is "none". That is, simply leaving the Office: component out of the StaffEmployee definition would result in an incomplete specification.

The definition of SalariedEmployee does not specify a type for "Office:", but this is acceptable since it is a class, and its instances can provide the required specialization. Hence, the Programmer components specify "Office: SharedOffice" and the Manager components specify "Office: PrivateOffice".

4.1. Multiple Inheritance

It is sometimes useful to have a single instance that inherits attributes from more than one parent. For example,

```
object MemberOfTechnicalStaff instance of Manager and Programmer is
    components: TechnicalSpeciality;
    description: (*
        A member of the technical staff is both a manager and
        a programmer.
    *);
end MemberOfTechnicalStaff;
```

The rule for multiple inheritance is that the instance inherits the *union* of the parent attributes. In particular, if the parents have one or more common attributes, then the instance has only a single version of the common attributes.

4.2. Restrictions on Inheritance and Specialization

Instances *inherit* components from a parent class, and can *add* new components to those that are inherited. Instances cannot however *uninherit* or *override* a component that is specified in the parent. Uninherit would mean that an instance could eliminate one or more parent components. Override would mean that if a parent component were specified with both a name and a type, that the type could be changed in the instance. Not all class-based languages have this additivity restriction, but RSL does.

4.3. Class Objects as Types

It was noted earlier that an abstract object formally defines a *type*, in the same sense as in strongly-typed programming languages. Using the class/subclass hierarchy allows the definition of *subtypes*. Specifically, an instance of an object class is considered a *subtype* of its parent class.

The major effect of subtyping in RSL relates to the use of class and instance objects in operation parameters. The typing rule for operation parameters is the following:

A formal parameter of a class type may accept an actual parameter of that type, as well as any instance type(s) of which the formal type is a parent.

Consider the following example:

```
object class Parent is ... ;
object class Child instance of Parent is ... ;
object class GrandChild instance of Child is ... ;

operation O is
    inputs: p:Parent, g:GrandChild;
    outputs: c:Child
```

Here Child is a subtype of Parent; GrandChild is a subtype of Child and in turn a subtype of Parent. Hence, by the typing rule above, the formal input parameter `p` can match an actual parameter of any type in the class hierarchy. I.e., an actual parameter of type Parent, Child, or GrandChild can be supplied to the input `p`. In contrast, only an actual parameter of type GrandChild can be supplied to the formal input `g`. The output parameter `c` produces a value of type Child or GrandChild, but not Parent.

Notice that subtyping of formal parameters is one-directional. That is, a formal parameter of a class type may accept actuals of itself or any instance type. However, a formal parameter of an instance type may *not* accept parameters of a parent type. This is the same general rule as in typed object-oriented programming languages, such as C++ [Stroustrup 91].

5. Modules

Entity definitions are packaged within *modules*. The syntax and semantics of RSL modules are similar to that of the Modula-2 programming language [Wirth 85]. The basic format of an RSL module is the following:

module *name* ;
  [*imports*]
  [*exports*]
  [*attribute-definitions* ... ;]
  [*entity definition*; |
  formal definition;] ...
end *name*

Module *imports* and *exports* optionally define inter-module name visibility. *Attribute-definitions* specify user-defined attributes, as described in Section 6 below. If any attribute definitions are present, they must appear before entity definitions. *Entity-definitions* are objects and operations, as described in the preceding section. *Formal-definitions* are discussed in Sections 7 and 8 below.

In terms of packaging, a module defines a *name scope* within which all defined entities are visible. Any definition within a given module may reference any other entity defined within the same module. Unlike many programming languages, an entity definition does not need to lexically precede its reference(s) within a module.

Normally, entities defined in two different modules are mutually invisible. For example if object A is defined in module M1 and operation B is defined in module M2, the definition of B cannot reference A as an input or output. The use of **import** and **export** declarations extends the visibility of names between modules. The general format of an *import* declaration is the following:

from *module-name* **import [qualified]** *entity-name*,...

and the format of **export** is:

export *entity-name*,...

Consider the following example:

```
module A;
    export O1;
    object O1 is ... ;
    object O2 is ... ;
end A;

module B;
    from A import O1;
    operation Op1 is
        inputs: O1, ... ;          (* Legal reference to O1 *)
    end Op1
    operation Op2 is
        inputs: O2, ... ;          (* Illegal reference to O2 *)
end A;
```

The import declaration in B makes object O1 visible within B. Hence, the reference to O1 in Op1 is fine. Since object O2 is not explicitly imported into B, the reference to O2 in Op2 is illegal.

It should be noted that imports must be matched by corresponding exports. That is, a name cannot be import into one module without having been export from another. Conversely, if a module exports one or more entities, each of these entities must be referenced by at least one import.

The use of import/export can lead to name conflicts if a module both imports and defines an entity of the same name. For example:

```
module A;
    export O1;
    object O1 is ... ;
end A;

module B
    from A import O1;
    object O1 is ... ;         (* Name conflict *)
    operation Op is
        inputs: O1;            (* Ambiguous reference *)
```

Here module B both imports and defines O1. To overcome such name conflicts, names can be import in *qualified* form, and referenced by prefixing with the name of the defining module. The following is a version of the immediately preceding example with the name conflict removed:

```
module A;
    export O1;
```

```
    object O1 is ... ;
end A;

module B
    from A import qualified O1;
    object O1 is ... ;          (* No conflict *)
    operation Op is
        inputs: O1;             (* Legal reference to B's O1 *)
        outputs: A.O1;          (* Legal reference to A's O1 *)
```

Note the use of the keyword **qualified** in the import clause. Here, reference to the imported version of O1 is denoted by the qualified reference "A.O1" within B. The unqualified reference to O1 refers to the O1 defined within B. Hence, there is no name conflict in this case, since both versions of O1 can be referenced unambiguously.

6. User-Defined Attributes

In the most general sense, the format of an entity definition is the following:

[object | operation] *name* **is**
attribute-name : *attribute-value*
...
end *name*

In the examples thus far, built-in attributes have been discussed. For an object, the built-in attributes are: **components**, **operations**, **equations**, and **description**. For an operation, built-in attributes are **components**, **inputs**, **outputs**, **preconditions**, **postconditions**, and **description**.

A user-defined attribute specifies additional relational or descriptive information about an object or operation. Consider the following example:

```
module M;
    define object attribute scheduled_by, coordinated_by, special_note;

    object Meeting is
        components: StartTime, EndTime, Attendees, ... ;
        scheduled_by: StaffPerson;
        coordinated_by: Manager;
        description: (* A meeting represents ... *);
        special_note: (* Ask our system analyst if this is correct *);
    end Meeting;

    object StaffPerson is ... ;
    object Manager is ... ;
end M
```

This example illustrates how attributes are defined and used. Before use, an attribute must be defined in a **define** clause of the following general form:

define [object | operation] attribute *attr-name*,...

Such a definition allows the listed *attr-names* to appear within entity definitions. In the example above, scheduled_by, coordinated_by, and special_note are defined object attributes. Note that object and operation attributes are separately declared, so if the same attribute is desired for both objects and operations, two separate **define** declarations must be given.

Once defined, an attribute can be put to two uses: (1) to define formal *relations* between entities -- a relational attribute; (2) to augment an entity definition with special-purpose comments -- a commentary attribute. These two uses are specified by the following two syntactic forms, respectively:

attr-name : *entity-name,...*

attr-name : *RSL comment*

In the Meeting example above, scheduled_by and coordinated_by are relational attributes; special_note is commentary.

A relation between entities specifies a non-hierarchical connection. To understand such connections, it is instructive to compare relational attributes to the built-in RSL component relation. Consider the following alternative to the Meeting specification above, where no relational attributes are used:

```
module M

    object Meeting is
        components: StartTime, EndTime, Attendees,
                        scheduled_by:StaffPerson, coordinated_by: Manager;
    end Meeting;

    object StaffPerson is ... ;
    object Manager is ... ;
end M
```

Here, what were formerly specified as relations are now *components* of the meeting. The relational versus hierarchical specifications define different conceptual views. In the relational specification, the scheduled_by and coordinated_by objects are not *part of* the Meeting as in the hierarchical definition. While the difference is subtle, it can be important in terms of constructing an accurate view of a system being specified.

The distinction between relational versus hierarchical specifications can be further clarified by considering bi-directional relations. Here is a further refinement of the Meeting example:

```
module M;
    define object attribute scheduled_by, coordinated_by, special_note;
    define object attribute scheduler_of, coordinator_of;

    object Meeting is
        components: StartTime, EndTime, Attendees, ... ;
        scheduled_by: StaffPerson;
        coordinated_by: Manager;
        description: (* A meeting represents ... *);
        special_note: (* Ask our system analyst if this is correct *);
    end Meeting;

    object StaffPerson is
        ...
        scheduler_of: Meeting
    end StaffPerson;

    object Manager is ... ;
        ...
        coordinator_of: Meeting
    end Manager;
end M
```

Here a bi-directional *scheduling* relation has been established between Meeting and StaffPerson. A similar bi-directional *coordinating* relation has been specified between Meeting and Manager. Such relations would be more awkward to specify using hierarchical components, and the hierarchy would be inappropriate in the sense that none of the objects participating in the relations is conceptually *part of* the others.

Commentary attributes are particularly useful to describe *meta*-properties of a requirements specification. The intention of the special_note attribute above is to supply a temporary annotation for use during the development of the specification. While it is possible to use the built-in **description** field for such commentary, it is clearly more awkward to do so, as the following version of Meeting illustrates:

```
module M;
    define object attribute scheduled_by, coordinated_by;

    object Meeting is
        components: StartTime, EndTime, Attendees, ... ;
        scheduled_by: StaffPerson;
        coordinated_by: Manager;
        description: (* A meeting represents ...
                SPECIAL_NOTE: Ask our system analyst if this is correct *);
    end Meeting;

    object StaffPerson is ... ;
    object Manager is ... ;
end M
```

It should be noted that there is a fully functional notation that can be substituted for the use of relational attributes. In this sense, relational attributes can be viewed as "syntactic sugar" for an equivalent functional definition. Details of the equivalence are discussed in Appendix C. Relational attributes are provided in RSL for specifiers who find relational notation conceptually convenient. Those users who would prefer to use a fully functional notation, while retaining the expression power of relations, should consult Appendix C.

7. Formal Specifications

RSL object and operation definitions can be augmented with formal mathematical specifications. These formal specifications are in three forms: *equational*, *predicative*, and *axiomatic*. Equational specifications are defined by associating a set of equations with an object. Predicative specifications are defined by associating preconditions and postconditions with an operation. Axiomatic specifications are defined by associating a set of global conditions with all of the objects and operations within a module.

The predominant form of logical expression used in a formal specification is the *predicate*. Semantically, a predicate is a mathematical formula with a boolean (i.e., true/false) value. A predicate is fundamentally the same form of boolean expression as found in programming languages. However, predicates in RSL can contain quantifiers and list operators that are typically unavailable in programming languages. In addition to boolean-valued expressions, RSL provides numeric, string, and list-valued expressions.

This section of the report is a terse syntactic description of the elements of a formal specification. Section 8 to follow contains a tutorial discussion of formal specification, with examples.

7.1. Variable names

The base elements in an expression are variable names. Variables are declared in name/type pairs that appear in component expressions, input/output lists, and other contexts. Syntactically, a variable name is an identifier.

The reader should note the distinction between a variable name, used in an expression, versus an object name. An object name is defined using a complete object definition. That is, object names are those and only those names defined in definitions starting "**object** *object-name* ... ;". In contrast, a variable name is defined only in the context of a name/type pair. Variable names are those and only those names defined as *variable-name:object-name*. Such name/type pairs appear in one of four syntactic contexts:

1. composition expressions appearing in the **components** part of an object definition
2. **input/output** lists in an operation definition
3. **equation** variable definitions
4. **forall** and **exists** clauses

The first two of these contexts were described in a preceding section of the report. The last two contexts are described shortly.

7.2. Functional, Arithmetic, and Boolean Expressions

A functional expression is the invocation of an operation or auxiliary function (auxiliary functions are defined below). The general form of a functional expression is:

name(args,...)

where *name* is the name of a defined operation or auxiliary function and *args* are input arguments. Any defined operation or auxiliary function can be used in a functional expression. The argument types of functional expressions are type checked in the same manner that function calls are type checked in a programming language. That is, the number and type of arguments must agree with the input/output declarations in the named operation.

The standard boolean operators **and**, **or**, and **not** are used in predicates. Note that **and** and **or** have overloaded meanings in RSL. In the context of a composition expression, **and** and **or** denote composition primitives. In the context of a predicate, **and** and **or** denote boolean operators.

Predicates can contain the relational operators =, ~= (not equal), <, >, <=, and >=. These are defined between expressions of the same type.

Predicate terms can include standard arithmetic operations on real numbers (including integers). The operators are +, -, *, /, **div**, and **mod**. Note that since a predicate must have a boolean value, arithmetic expressions can only be used in predicates in the context of other logical operations, such as comparisons. For example, "a+b" is a legal expression but not a legal predicate. "(a+b) > 10" is a legal predicate.

7.3. List Operations

List operations available for use in predicates are: || (concatenation) # (length), and **in** (element of). List operations can be used on variables denoting a list-composed object.

7.4. Composite Object Selector Operations

When an object is declared with multiple **and**ed subcomponents, there is a built-in **and** selector operation for each such component, as the following general definition illustrates:

```
object Foo is
   components: Foo1 and Foo2 and ... and Foon;
```

```
   (*Built-in operations:*)
        SelectFoo1: (Foo) -> (Foo1);
        SelectFoo2: (Foo) -> (Foo2);
              . . .
        SelectFoon: (Foo) -> (Foon);
end Foo;
```

An alternate, ''syntactically sugared'' notation for **and**-selector operations is the infix "." operator. Using infix ".", the following two terms are the same:

`SelectFoo1(foo)` *is equivalent to* `foo.Foo1`

Note that the "." operation is analogous to the use of "." in a programming language. Namely, "." selects an and-component in the same manner that "." selects a record field in a programming language.

When an object is declared with multiple **or**'d subcomponents, there exist two **or**-selector operations for each such component, as the following general definition illustrates:

```
object Foo is
   components: Foo1 or Foo2 or ... or Foon

   (* assumable operations: *)
        IsFoo1: (Foo) -> (bool);
        IsFoo2: (Foo) -> (bool);
                  . . .
        IsFoon: (Foo) -> (bool);
        SelectFoo1: (Foo) -> (Foo1);
        SelectFoo2: (Foo) -> (Foo2);
                  . . .
        SelectFoon: (Foo) -> (Foon);
end Foo;
```

The operations prefixed with "Is" are used to determine which alternative a subobject is. The operations prefixed with "Select" are used in the same fashion as the selectors for an and-composed object.

An alternate, "syntactically sugared" notation for **or**-selector operations is the infix "?" operator. Using infix "?", the following two terms are the same:

`IsFoo1(Foo)` *is equivalent to* `Foo?Foo1`

A common use for the or-selector is in objects with an error alternative. Consider the following example:

```
object ValueOrError is
    components: Value or Error;
end ValueOrError;
```

for which ValueOrError?Value is true if the subcomponent of foo is Value, or ValueOrError?Error is true if the subcomponent is an Error subobject. The "?" operator allows or'd components to be "safely" accessed. That is, before accessing an or component via ".", it should be checked with "?".

When an object is declared with a *list* of subcomponents, there is a built-in **list** selector operation for an individual subcomponent and a sublist operation that selects a range of subcomponents. The following general definition illustrates these list selectors:

```
object Foo is
    components: Foo1*;
    (*Built-in operations:*)
        SelectNthFoo1: (Foo,number) -> (Foo1);
```

```
        SelectMthruNthFoo: (Foo,number,number) -> (Foo);
end Foo
```

An alternate, ''syntactically sugared'' notation for **list**-selector uses "[" and "]" brackets. Using brackets, the following two terms are the same:

`SelectNthFoo1(foo,n)` *is equivalent to* `foo[n]`

as are the following two:

`SelectMthruNthFoo(foo,m,n)` *is equivalent to* `foo[m:n]`

Note that the "[...]" operation is analogous to the use of "[...]" in a programming language. Namely, "O[n]" selects the nth component of a list-composed object in the same way that "A[n]" selects the nth component of an array in a programming language.

In relation to other predicate operators, ".", "?", and "[...]" have the highest precedence.

7.5. Logical Implication and Quantifiers

Logical implication operations are **if-then-else**, "=>" (**implies**), and "<=>" (**iff**). Note that this **if-then-else** is a boolean operator, not a control construct as in a programming language. In the expression

if T **then** P1 **else** P2

T is a predicate, P1 and P2 are each expressions, and the value of the expression is P1 if T is true, or P2 if T is false.

Universal and existential quantifiers can appear in predicates. In normal mathematical notation, universal quantification is represented by an upside down "A" and existential quantification by a backwards "E". In RSL, the quantifier operators are **forall** and **exists**. The general form of universal quantification is

forall *(x:t) predicate*

read as "for all values *x* of type *t*, *predicate* is true" where *x* must appear somewhere in *predicate*. The general form of existential quantification in RSL is

exists *(x:t) predicate*

read as "there exists an *x* of type *t* such that *predicate* is true" where *x* must appear somewhere in *predicate*.

In standard mathematical logic, universal quantification typically takes one of the following two syntactic forms:

$$(1)\quad \forall\,(x \mid p1(x))\; p2(x)$$
$$(2)\quad \forall\,(x \in S)\; p(x)$$

The reading of form (1) is "for all values x such that predicate p1(x) is true, p2(x) is also true". The reading of form (2) is "for all elements x in set S, predicate p(x) is true"

In RSL, the forall operator quantifies over all values of a particular object type. In comparison to forms (1) and (2) above, the general form of universal quantification in RSL is

```
forall (x: O) p(x)
```

The reading of this RSL form of quantification is "for all values x of object type O, predicate p(x) is true". The reason that RSL quantifies over object types is that RSL is based on *typed logic*. This means that all variables that appear in RSL predicates must be of some object type. In many mathematical treatments of quantification, the issue of strong value typing may not arise, and hence the notion of quantifying over types does not arise.

The fact that RSL is based on typed logic does not restrict how universal quantification can be used, it just means that the use of universal quantification in RSL must take typing into account. Therefore, the specific format of quantification is slightly different than in untyped logics.

Both quantification form (1) and form (2) above can be easily represented in RSL. The RSL for form (1) is:

```
forall (x:SomeObject) if p1(x) then p2(x)
```

where the predicates operations p1 and p2 take SomeObject as input and output boolean. The RSL notation for quantification form (2) is:

```
forall (x:Elem) if x in S then p(x)
```

where object S must be composed of Elem*. Since these two forms of quantification are quite typical, RSL provides alternative syntactic notation to express them conveniently. Specifically the following two RSL forms correspond to universal quantification forms (1) and (2) above:

(1) **forall** *(x:t | predicate) predicate*
(2) **forall** *(x in list) predicate*

In form (2), *list* must be a list-composed object, and x will be of the component type of *list*. For example,

```
object SomeSet is
    components: Elem*;
    . . .
end SomeSet;

operation SomeSetOp is
    inputs: s:SomeSet;
    outputs: s':SomeSet;
    postcond: forall (e in S) f(e);
end SomeSetOp;
```

In this example, `e` is a variable of type Elem within the body of the `forall`.

Section 8 below contains further examples on the use of universal and existential quantification in RSL.

7.6. Pre/Postconditions, Equations, and Axioms

Preconditions and postconditions are associated with an operation by including precondition and/or postcondition declarations within the operation definition. The syntactic forms are:

precondition: *predicate* ;
postcondition: *predicate* ;

where *predicate* is a legal RSL predicate expression. Typically, conditions are made up of a number of predicates, composed with boolean "and's" and "or's". Section 8 has a number of examples that illustrate the use of pre/postconditions in RSL.

Equations are associated with an object by including an equation declaration within the object definition. The general form of equation definition is the following:

equations:
  var *var_name:object_name, ...* ;
  functional_expr == quantifier_free_expression;
    ... ;

where a *functional_expr* is as defined above, and a *quantifier_free_expression* is an expression that contains no quantifiers or list operations. Note that the quantifier_free_expression is not limited to a boolean value, that is, it need not be only a predicate. The variable declaration(s) that precede the equations define auxiliary variables that are used in the equations. The next section of the report contains example equation definitions in RSL.

The "==" operator and the "=" operator should not be confused. The "==" separates the two sides of an equation; it has the lowest precedence of any infix operator. The "=" is the logical equality operator, which returns a boolean value. It has the same precedence as the other comparison operators.

Axioms are associated with all of the objects and operations defined within a module. Syntactically, an axiom is a predicate:

axiom: *predicate* ;

7.7. Auxiliary Functions

As noted in Section 1.3, the objects and operations in an RSL specification should be those that are visible to end-users of the specified system. When a specification is fully formalized, it is sometimes necessary or convenient to define *auxiliary* functions that are referenced in pre/postconditions or equations. An auxiliary function differs from an operation in that it is not intended to be visible to the end user of the specified system.

Consider the following example:

```
object NumericPair is
    components: n1:Number, n2:Number;
end NumericPair;

operation DoSomethingWithPairs is
    inputs: p1: NumericPair, p2: NumericPair, s: SomeOtherObject;
    output: s':SomeOtherObject;
    precondition: if PairwiseLess(p1,p2) then ... ;
end DoSomethingWithPairs;

function PairwiseLess(p1:NumericPair, p2:NumericPair) : (boolean) =
      (p1.n1 < p2.n1) and (p1.n2 < p2.n2);
```

Here the auxiliary function PairwiseLess is defined. This function takes inputs p1 and p2, both of type NumericPair. It produces a boolean output. The following is the general format of an auxiliary function:

function *name* (*list of inputs*) : (*list of outputs*) =
  body
end *name*

The function *body* is an expression of the output type(s). If the function produces a single output, then the body is an expression of that type. If the function produces more than one value, then the body is a list of expressions enclosed in square brackets, where each expression corresponds ordinally by type to each of the output parameters.

While PairwiseLess is not strictly necessary in the preceding example, it can make the specification clearer and more concise, particularly if PairwiseLess is used in several different preconditions and postconditions. The next section contains further examples on the use of auxiliary functions and a discussion of when their use is appropriate.

8. Formal Specification Examples

The four object composition primitives in RSL are just that -- *primitive*. In particular, using just these four primitives, there is no way to specify any of a number of important properties about objects and operations. For example, in the PersonDatabase object suppose it is necessary to specify that the database cannot have duplicate entries. Or, suppose it is necessary that the database behave as a queue-like structure, such that new records are added on a first-come-first-served (FIFO) basis. Neither of these or similar properties can be defined with the use of composition primitives alone. The use of formal specification is necessary in such cases.

One method to describe properties such as these is to use English. It is in fact useful to describe all properties of an entity in its English description. However, it is not sufficient to rely solely on natural language to specify critical properties. To do so soon leads to well-known problems of ambiguity and imprecision. Therefore, it is necessary to use the more formal language of mathematics to obtain a truly precise and unambiguous specification.

Another method to describe properties is to use a computer programming language. This has the advantage of being fully precise and unambiguous. However, using a programming language for specification is fundamentally contrary to the purpose of specification. Namely, a specification should be as free as possible of implementation detail. The purpose of a programming language is precisely for the expression of implementation detail.

To overcome the disadvantages of English and programming languages, researchers have developed a number of formal techniques expressly for use in specification. The two formal techniques supported primarily in RSL are *algebraic* and *predicative* specification. With the algebraic technique, formal properties are specified as a set of equations associated with an object. Hence, algebraic specifications can be considered *object-oriented*. In the predicative technique, formal properties are specified as preconditions and postconditions on operations. Hence, predicative specification can be considered *operation-oriented*.

It is important to note that the two techniques provide different approaches to formal specification. A system can be fully specified using only the algebraic approach, it can be fully specified using only the predicative approach, or it can be specified using a combination of the two approaches.

A potential problem with either of these forms of specification is the introduction of *implementation biases* into a specification. As examples below illustrate, these formal specification techniques tread a fine line between specification and implementation. In fact, what many software engineers call a *specification language* others may call a *very high-level programming language*.

The gist of the problem is that the more precise we try to become with a specification, the more we tend to constrain what the implementation can look like. However, this violates the general principle that a specification is as free from implementation details as possible.

Another problem is that in some cases we find that in order to state a specification sufficiently precisely, we need to interject *auxiliary functions*, thereby violating the general rule that a specification should contain only entities directly visible to the end user.

In summary, specification is a continual battle between risking imprecision by saying too little versus risking the addition of implementation details by saying too much. The experienced specifier learns how to wage this battle successfully.

8.1. Equational Specification

An equational specification formally defines an object in terms of equations between the operations of the object. Equational specifications are desirable because they require no underlying data model. Rather, the definition of an object is stated entirely in terms of its abstract operations. This model-free property of equational specification is generally not satisfied by other forms of formal specification. For example, the predicative style of specification defined in the next section of the report is not model-free. Rather, predicative specification relies on a list-based data model to provide a basis for defining formal preconditions and postconditions. Without reliance on some underlying model, the predicative style of specification would not be complete.

The process of defining an equational specification is divided into three major steps:

1. Define the *operation signatures* for the object's operations
2. Categorize the operations into *constructors*, *destructors*, *selectors*, and *initializers*
3. Define the equations

To illustrate these steps, consider the definition of a database object that has a set-like structure. That is, it has the property that duplicate entries are not allowed. Here is an equational definition for such a database:

```
object SetDB is
    components: Elem*;      (* Note the minimal representation *)

    operations:
        Insert: (SetDB, Elem) -> (SetDB),    (* Constructor operation *)
        Delete: (SetDB, Elem) -> (SetDB),    (* Destructor operation *)
        Find: (SetDB, Elem) -> (boolean),    (* Selector operation *)
        EmptyDB: () -> (SetDB);              (* Initializer operation *)

    equations:
        var s: SetDB; e, e': Elem;

        Find(EmptyDB()) == false;
        Find(Insert(s, e), e') ==
            if e=e' then true else Find(s, e');
        Delete(EmptyDB(), e) == EmptyDB();
        Delete(Insert(s, e), e') ==
            if e=e'
            then Delete(s, e')
            else Insert(Delete(s, e'), e);

end SetDB;
```

The first two lines of this definition use the standard RSL notation for defining any object. When defining an object equationally, its component structure should always be an expression of the form "C*", where C is a single object name. In the case of the SetDB, the components are Elem*, where Elem is some type of element defined elsewhere.

The operations section of the SetDB definition contains the full signatures for its operations. The term *signature* refers to the inputs and outputs of an operation. The general form of an operation signature in RSL is

operation name : (*list of inputs*) -> (*list of outputs*) where the input and output lists contain zero or more object names.

Before defining the equations, an object's operations are organized into the following categories:

1. *Constructor* and *initializer* operations that build an object out of smaller parts. These two categories can be grouped under the common heading of *constructors*.
2. *Selector* and *destructor* operations that access or remove some component of an object. These two categories can be grouped under the common heading of *selectors*.
3. *Derived* operations that can be defined formally in terms of one or more constructors or selectors.

The comments in the SetDB definition indicate the categories of each of the operations.

The important characteristic of constructor operations is that they are *additive*. That is, the resulting output of a constructor is a combination of the inputs. In contrast, selector operations are *subtractive*. That is, the resulting output is some smaller piece of the input. In general, an equationally defined object should have at least one each of the constructor, initializer, destructor, and selector operations.

Once operations have been fully defined and categorized, the equations themselves are defined. The general format of an equation is the following:

functional_expr == quantifier_free_expression;

where a *functional_expr* is defined above, and a *quantifier_free_expression* is an expression that contains no quantifiers or set operations. The "==" operator separates an equation into a left-hand side (LHS) and a right-hand side (RHS).

The following guidelines are useful for constructing a correct equational specification:

a. The equations define what the selector operations do to the constructor operations, and not vice versa. That is, the outermost operation name in the LHS of an equation should always be a selector, not a constructor.

b. There should be one equation that defines what each selector does to each constructor. Hence, if there are c constructors and s selectors, then there are $c * s$ equations.

c. Equations are defined in an *inductive style*. Specifically, equations are defined in groups, where the first of the group specifies what a selector does with an initializer. The remaining equations in the inductive group specify what a selector does with the other constructors.

Consider how these guidelines were used to construct the SetDB equations:

a. The SetDB selectors are Find and Delete. Hence, each of the SetDB equations has one of these as the outermost operation in the LHS of the equation.

b. Since there are 2 selectors and 2 constructors, there are a total of 4 equations.

c. The inductive style of definition indicates that two groups of equations are appropriate. The first group of two equations specifies what Find does to EmptyDB and Insert. The second group of equations specifies what Delete does to EmptyDB and Insert.

The most difficult part of an equational definition is constructing the second part of the inductive definition. That is, defining the equations for what the selectors do with the non-initializing constructors. To examine this part of process, let us review what an inductive style definition is.

To begin, it is assumed that the reader is familiar with the concept of proof by mathematical induction. Readers entirely unfamiliar with this topic should consult an appropriate reference. Recall that an inductive proof is a two step process, in which a proposition p(x) is proved for all values of x. The steps are:

1. Proof of the base case: set x=0 and prove p(x).
2. Proof of the inductive step: assume p(x) is true for x=n and then prove that p(x) is true for x=n+1.

An inductive equational definition involves a similar two-step process. That is, we define one equation for a base case and one for an inductive step. In the base-case equation, there is an analog to setting the induction variable to 0 -- we define the equation for an object that has 0 components. That is, the base case equation defines what a selector does to an initializer, since an initializer constructs an object with zero components. In the SetDB example, the base case equations are:

```
Find(EmptyDB()) == false;
```

and

```
Delete(EmptyDB(), e) == EmptyDB();
```

These equations define what the selectors do with a SetDB of size 0, that is a SetDB constructed with the EmptyDB initializer. Both these equations are quite intuitive -- if we try to find something in a SetDB of size 0, we receive a false result back. If we try to delete something from a SetDB of size 0, we get back the same 0-sized SetDB we started with. As in a mathematical induction proof, the base case is generally easy to establish.

The two inductive step equations in the SetDB example are:

```
Find(Insert(s, e), e') ==
    if e=e' then true else Find(s, e');
```

and

```
Delete(Insert(s, e), e') ==
    if e=e'
    then Delete(s, e')
    else Insert(Delete(s, e'), e);
```

These equations specify what the selectors do with a SetDB of size n+1. The derivation of n+1 comes from an assumption about the set *s* that is the first argument to Insert, Find, and Delete. The assumption is that *s* contains *n* elements. Hence, the constructed object *Insert(s, e)* contains *n+1* elements -- the *n* elements assumed to be in *s* plus the one new element *e* that is inserted.

Both of the inductive-step equations use a recursive style of definition. This is very common in inductive-step equations. The recursion always involves some form of conditional, typically an if-then-else. The test of the condition checks the *n+1* case, then the recursion handles the other *n* cases. Consider again the case for Find:

```
Find(Insert(s, e), e') ==
    if e=e' then true else Find(s, e');
```

To paraphrase this equation in English, the LHS says "consider what the Find selector does with a SetDB of size n+1". The RHS then says, "if the last item that was inserted is the item we're trying to find, then we've found it so return true; otherwise, the item we're looking for, if present, must be one of the n items in the SetDB s, so we recursively apply Find to look for it."

By far the most difficult equation in the SetDB example is the last one that defines the inductive step for Delete. The equation is not necessarily intuitive in terms of exactly how it defines set-like behavior. An important technique to determine if a set of equations does what we want is to formally *test* the equations. To gain an understanding of how the last equation works, we will apply this testing technique shortly. Before doing so, we examine some additional equational definitions.

8.2. Some Additional Equational Definitions

The SetDB example defined equations for the database property that duplicate entries are not allowed. Given below are additional equational definitions that specify other database structural properties. These are all properties that are considered appropriate to be known by an end user.

8.2.1. A Bag-Like Object

Suppose the desired definition of database is one that *does* allow duplicate entries, that is, it behaves like a mathematical *bag* instead of a set. To obtain the definition of a bag from that of a set, only one change is necessary -- the **then** clause of the last SetDB equation is changed from

```
then Delete(s, e')    (* for a set *)
```

to

```
then s                (* for a bag *)
```

The upcoming section on testing equations will discuss why this change does what is claimed.

8.2.2. A LIFO-Structured Database

Suppose we choose to define a database with a LIFO (Last-In, First-Out) structure, that is, a database that behaves like a stack. A stack is much simpler than a set since no recursive searching is necessary for the selector operation. For a stack, the selector is the Top operation. Here is the equational definition:

```
object StackDB is
    components: Elem*;

    operations:
        Push: (StackDB, Elem) -> (StackDB),  (* constructor operation *)
        Pop: (StackDB) -> (StackDB),         (* destructor operation *)
        Top: (StackDB) -> (Elem),            (* selector operation *)
        EmptyStackDB: () -> (StackDB);       (* initializer operation *)

    equations:
        var s: StackDB; e: Elem;

        Top(EmptyStackDB()) == EmptyElem();
        Top(Push(s, e)) == e;
        Pop(EmptyStackDB) == EmptyStackDB();
        Pop(Push(s, e)) == s;

end StackDB;
```

8.2.3. A FIFO-Structured Database

Next we consider the definition of a DB with a FIFO (First-In, First-Out) property, that is, the definition of a queue-like DB. This definition is intermediate in terms of complexity between a SetDB and a StackDB.

```
object QueueDB
    components: Elem*;

    operations:
        Enq: (QueueDB, Elem) -> (QueueDB),   (* constructor operation *)
        Deq: (QueueDB) -> (QueueDB),         (* destructor operation *)
        Front: (QueueDB) -> (Elem),          (* selector operation *)
        EmptyQueueDB: () -> (QueueDB);       (* initializer operation *)

    equations:
        var q: QueueDB; e:Elem;

        Front(EmptyQueueDB()) == EmtpyElem();
        Front(Enq(q, e)) ==
            if q = EmptyQueueDB()
            then EmptyQueueDB()
            else Front(q);
        Deq(EmptyQueueDB) == EmptyQueueDB();
        Deq(Enq(q, e)) ==
            if q = EmptyQueueDB
            then EmptyQueueDB
            else Enq(Deq(q), e);

end QueueDB;
```

8.2.4. A Keyed Database

Finally, we consider the definition of a keyed database, in which duplicates are not allowed. The difference between this and SetDB is a more realistic version of the Find operation. No real database would have Find return a boolean. Rather, Find should return a whole element, which would be located by some Key. The following specification defines this form of database. The equations have the same fundamental structure as SetDB, but with the addition of a Key argument where appropriate.

```
object DB is
    components: Elem*;

    operations:
        Insert: (DB, Key, Elem) -> DB;   (* constructor operation *)
        Delete: (DB, Key) -> (DB);       (* destructor operation *)
        Find: (DB, Key) -> (Elem);       (* selector operation *)
        EmptyDB: () -> (DB);             (* initializer operation *)

    equations:
        var d: DB; e: Elem; k,k': Key;
        Find(EmptyDB, k) == EmptyElem();
        Find(Insert(d, e, k), k') ==
            if k=k' then e else find(d, k');
        Delete(EmptyDB, k) == EmptyDB;
        Delete(Insert(d, e, k), k') ==
            if k=k'
```

```
                then Delete(d, k')
                else Insert(Delete(d, k'), k, e);
end DB;
```

8.3. Testing Equational Specifications

The first step in testing an equational specification is to understand precisely how equationally defined objects are represented. Since an equationally-defined object has no concrete representation, we have no real way to "get our hands on it." We cannot draw a picture of an object that has no concrete representation. The only means to represent such an object is by using its *constructor operations*.

Consider the representation of SetDB objects. To build a SetDB, the component type Elem must be defined. Assume for simplicity object Elem is defined as a number. Given this, a SetDB object will contain numeric values. As a concrete example, the SetDB containing elements 1, 2, and 3 is represented as a functional expression containing three applications of the Insert constructor:

```
Insert(Insert(Insert(EmtpyDB(), 1), 2), 3)
```

In general, any equationally-defined object can be represented as a sequence of constructor operations, applied at the base to an initializer operation.

When a selector operation is applied to a constructed object, the constructed object simply appears as an argument in the appropriate selector argument position. For example, to delete the element 2 for the above set, the Delete operation is applied as follows:

```
Delete(Insert(Insert(Insert(EmtpyDB(), 1), 2), 3), 2)
```

Functional expressions such as this are commonly called "terms". A term is any number of equationally-defined operations, applied properly according to their signature definitions.

The general method to test a set of equations is to consider the equations as *reduction rules* and *apply* these rules to a constructed object. In general, a reduction rule is of the form

LHS $\rightarrow$ RHS

where the "$\rightarrow$" symbol is read "reduces to". To treat an equation as a rewrite rule, the "==" is simply replaced by a "$\rightarrow$". We do not *actually* change the "==", but rather we *view* the equation as a rewrite rule, as if the "==" were "$\rightarrow$".

A reduction rule is said to be *applied* to some *subject*. In the case of an equational rewrite rule, the subject is a term. Rewrite rule application involves *matching* the LHS of the rule to some part of the term, and then replacing the matched term with the RHS of the rule. This replacement of the matched LHS with a RHS is the actual *rewriting* process.

The goal of equational term reduction is to remove all selector operations from a term, leaving only constructors. This goal makes sense when we consider the form of the equations. Recall that equations are written in terms of what a selector does to a constructor, not the other way around. The reason that we do not define equations for what constructors do is that they need not be reduced. In other words, a term containing only constructors is considered fully reduced. In this way, an actual object is represented fundamentally by a series of constructor operations. Whenever a selector is applied to a constructed object, the selector is "reduced out" to produce a term that again only contains constructors.

Consider the reduction of the Delete term just above:

```
Delete(Insert(Insert(Insert(EmtpyDB(), 1), 2), 3), 2)
```

The goal of the reduction is the following object

```
(Insert(Insert(EmtpyDB(), 1), 3)
```

which represents a SetDB containing a 1 and 3 -- the 2 has been deleted as expected. This goal was reached by a series of rule applications that matched an equation LHS to some part of the term, and then replaced the matched LHS with the RHS of the equation. Let us trace through the reduction steps for the above term. We first start with the subject term:

```
Delete(Insert(Insert(Insert(EmtpyDB(), 1), 2), 3), 2)
```

To find a match, we consult the SetDB equations, looking for a LHS match. The process we apply is *pattern matching*. That is, we consider the LHS of each equation to be a pattern that we attempt to match to some part of the term. The matching technique attempts to match each operation name in a LHS with the same name in the term. The variables in the LHS are matched to components in the term of the appropriate type.

Consider how the match is attempted on the above term with each of the four SetDB equations. The LHSs of the first two equations contain a Find operation. These two equations can be eliminated from consideration immediately, since there is no occurrence of Find anywhere in the subject term (i.e., there are only Delete, Insert, and EmptyDB). The LHS of the last two SetDB equations are possible candidates for a match, since they start with Delete. The third equation cannot match however, since there is no pattern in the subject term that will match.

```
Delete(EmptyDB(), ...)
```

This is because the only occurrence of Delete in the subject term is applied to Insert, not EmptyDB, and the operation names "Insert" and "EmptyDB" do not match.

We are left with the fourth SetDB equation. This equation does in fact match, as follows:

LHS Subterm	Matched Subject Subterm
`Delete`	`Delete`
`Insert`	outermost application of `Insert`
`s`	`Insert(Insert(EmptyDB(), 1), 2)`
`e`	3
`e'`	2

where a *subterm* is some part of a term. Given this matching, the rule can now be applied, i.e., *reduced*. The reduction involves systematic substitution of the matched LHS with the appropriate subterms of the RHS of equation 4. The RHS of equation 4 is the following:

```
if e=e'
then Delete(s, e')
else Insert(Delete(s, e'), e);
```

When the RHS of an equation contains an if-then-else, the replacement will be selected from either the then clause or the else clause. This requires the the evaluation of the if predicate. In this case, the if predicate is

```
e = e'
```

Based on the above subterm matches, this predicate evaluates to

```
3 = 2
```

which is false. Hence, the RHS replacement is the term given in the else clause of the if-then-else, which is

```
Insert(Delete(s, e'), e);
```

That is, we must substitute this term for the matched LHS. Doing this, the first step of the overall reduction process yields the following results:

```
Delete(Insert(Insert(Insert(EmptyDB, 1), 2), 3), 2)   =>
    Insert(Delete(Insert(Insert(EmptyDB, 1), 2), 2), 3)
```

Notice what has happened here: the Delete has been moved inward in the term passed the Insert. Also, the 2 has been moved inward passed the 3. This sort of manipulation is the gist of what term writing accomplishes. Namely, the term is rewritten by (re)moving pieces according to the equations.

Since the term above still contains a selector operation, it is not yet fully reduced. We therefore apply the same matching process as was used for the first reduction step. As in that case, the first two equations are eliminated immediately as possible matches. The LHS of the third equation also fails to match as before. We again match the fourth equation, this time as follows:

LHS Subterm	Matched Subject Subterm
`Delete`	`Delete`
`Insert`	2nd to the outermost application of `Insert`
`s`	`Insert(Insert(EmptyDB(), 1), 2)`
`e`	2
`e'`	2

The difference in this case is that the evaluation of the if predicate in the RHS of equation 4 is now

```
2 = 2
```

which evaluates to true this time. Hence, now instead of the else clause, we substitute the then clause, which is:

```
then Delete(s, e')
```

Doing this, the second step of the reduction process yields the following results:

```
Insert(Delete(Insert(Insert(EmptyDB, 1), 2), 1), 2) =>
    Insert(Delete(Insert(EmptyDB(), 1), 2), 3)
```

The resulting term is not yet fully reduced, since it still contains the Delete selector. Two additional reduction steps will occur to arrive at complete reduction. The next step will match equation 4 again, as in the preceding two steps. On this match, the if predicate, e=e', will test

```
1 = 2
```

which is false, thereby causing substitution of the else clause. Doing this, the third step of the reduction yields:

```
Insert(Delete(Insert(EmptyDB(), 1), 2), 3) =>
    Insert(Insert(Delete(EmptyDB(), 1), 2), 3)
```

The final step of the reduction uses SetDB equation 3. This is because the pattern matching finds "... Delete(EmptyDB() ..." in the term, which matches the LHS of equation 3. Given this, the final step of the reduction is

```
Insert(Insert(Delete(EmptyDB(), 1), 2), 3) =>
    Insert(Insert(EmptyDB(), 1), 3)
```

In addition to testing that SetDB equations are correct, the above reduction sequence reveals precisely how the SetDB equations work. It should now be clear why equation 4 of SetDB is written as it is. Namely, it is up to the Delete equation to make sure that the set property is maintained. Any number of Insert operations can be applied to a SetDB term, including Insert's of the same element. While this temporarily violates the set property, the Delete equation makes sure that set behavior is maintained by deleting *all* of the same elements that may have been inserted. While this may seem to be an inefficient manner in which to maintain the set property, we do not care about efficiency at all, only that the desired property is properly maintained.

8.4. Predicative Specification

Predicative specification is a complementary form of specification to equational. In the predicative approach, preconditions and postconditions are associated with an operation. The precondition specifies a predicate that must be true before the operation begins. The postcondition specifies a predicate that must be true after the operation completes.

Consider an alternate definition of a set-like database:

```
object SetDB is
    components:  Elem*;
    operations: Insert, Delete, Find, EmptyDB;

    (* Note that we do not need full operation signatures, nor
     * equations, since we are now specifying with pre and post-
     * conditions in place of the equations we used above.
     *)

end SetDB;

operation Insert is
    inputs: d:SetDB, e:Elem;
    outputs: d':SetDB;
    precondition: Find(d, e) = false;
         (* An equivalent precondition is: not (s in e);
          * see discussion below. *)
    postcondition: Find(d, e) = true;
         (* An equivalent precondition is: s in e;
          * see discussion below. *)
end Insert;

operation Find is
    inputs: d:SetDB, e:Elem;
    outputs: b:boolean;
    precondition: (* An empty precond means true *) ;
    postcondition: b = e in d;
end Find;

operation Delete is
    inputs: d:SetDB, e:Elem;
    outputs: d':SetDB;
    precondition: e in d;
         (* See discussion below about removing this precond. *)
    postcondition: not Find(d,e);
end Delete;

operation EmptyDB is
    inputs: ;
    outputs: d:SetDB;
    precondition: ;
```

```
    postcondition: forall (e:Elem) not Find(d,e);
        (* See discussion below about removing this precond. *)
end EmptyDB;
```

It is instructive to compare and contrast this predicative definition of SetDB with the equational definition given in the preceding section. Both definitions specify the same meaning in different forms. The following are some important points about specification with pre and post conditions in particular.

By definition, violation of a precondition is an error. Abstractly this means simply that the operation fails and no postcondition happens. Concretely (i.e., in the user's manual), this means that the end-user should see some form of appropriate error message.

A specification can be *weakened* or *strengthened* by the selective removal or addition of pre and/or post conditions. E.g.,

```
op Delete is
    in: d:DB, e:Elem;
    out: d':DB;
    precond: Find(d, e) = true;
    postcond: Find(d', e) = false;
end Delete;
```

is relatively weaker than

```
op Delete is
    in: d:DB, e:Elem;
    out: d':DB;
    postcond: Find(d', e) = false;
end Delete;
```

in that the latter specification says that it is an error to try to delete an element that is not already in the database, whereas the former is non-committal.

8.4.1. Quantification

In equational specification, recursion is used to specify a single equation that can be applied in an indefinite (potentially infinite) number of cases. That is, a recursive equation can be applicable to objects from size 1 to an infinite size. The predicative style of specification uses *quantification* to define predicates that apply to objects of an indefinite size.

Consider the following basic example of universal quantification:

```
operation FindAllYoungFolks is
    inputs: pdb: PersonDatabase;
    outputs: nl: NameList;
    postcondition:
        forall (p: PersonRecord)
            if (p in pdb) and (p.a < 40)
            then p.n in nl;
    description: (*
        FindAllYoungFolks produces a list of the names of all
        persons in the PersonDatabase whose age is less than 40
    *);
end FindAllYoungFolks;

object PersonDatabase is
    components: PersonRecord*;
    description: (* Same as earlier definitions. *)
end PersonDatabase;
```

```
object PersonRecord is
    components: n:Name and a:Age and ad:Address;
    description: (* Same as earlier definitions. *)
end PersonRecord;

object NameList is
    components: Name*;
end NameList;
```

The postcondition of the FindAllYoungFolks operation uses **forall** to quantify over the input PersonDatabase. An English paraphrase of the postcondition is as follows: " For each Person-Record, p, in the input, if the age of person p is less than 40, then the name of person p is in the output name list". This form of quantification is very typical of postconditions on operations that produce list-structured objects as outputs.

The following objects and operations further exemplify the use of universal and existential quantification in RSL. Consider an operation that merges set-like databases, of the type specified earlier:

```
operation MergeDBs
    inputs: d1:SetDB, d2:SetDB;
    outputs: d3:SetDB;
    postcondition:
        forall (e: Elem)
            if (e in d1) or (e in d2)
            then e in d3;
    description: (*
        The MergeDBs operation merges two databases of the same type
        of element.  The postcondition states that the result of the
        merge is that any element that is in either input d1 or d2
        must be in the output d3.
    *)
end MergeDBs;
```

There is no precondition needed. The postcondition states that if an element is in either of the input DBs, then it is in the output DB. An interesting question is the strength of this postcondition. In particular, does it guarantee that there are no duplicates in the output d3? Since the "in" operator is not constructive, based on this postcondition alone there is no way to state for certain whether d3 has duplicates or not. That is, this postcondition is too weak to guarantee no duplicates. Thus, we must look elsewhere.

The "elsewhere" we look is in the specification of the other DB operations. Specifically, the specifications for SetDB Find and Delete are:

```
operation Find is
    inputs: d:SetDB, e:Elem;
    outputs: b:boolean;
    precondition: (* An empty precond means true *) ;
    postcondition: b = e in d;
end Find;

operation Delete is
    inputs: d:SetDB, e:Elem;
    outputs: d':SetDB;
    precondition: e in d;
         (* See discussion below about removing this precond. *)
    postcondition: not Find(d,e);
end Delete;
```

What Find ensures is that if at least one copy of an element is in a SetDB, then it can be found. What Delete ensures, is if there are one or more copies of an element in a SetDB, then after the Delete none can be found. Therefore, whether duplicates are ever *physically present* in a SetDB is immaterial. The specifications of Find and Delete uphold the critical set property that if an element has been added at least once, then it can be found, and if it is deleted, it cannot be found. Hence, even if MergeDBs does add duplicates, Find and Delete will maintain the set property when subsequently invoked.

Consider another form of database merge:

```
object PairedDB is
    components: ElemPair*;
end PairedDB;

object ElemPair is
    components: e1:Elem and e2:Elem;
    description: (* An ElemPair is just a pair of elements. *)
end ElemPair;

operation PairwiseMergeDBs
    inputs: d1:DB, d2:DB;
    outputs: dp: PairedDB
    precond: #d
    postcondition:
        forall (e1,e2: Elem)
            if (e1 in d1) and (e2 in d2)
            then
                exists (ep: ElemPair) (ep in dp) and
                           (ep.e1 = e1) and (ep.e2 = e2)
    description: (*
        PairwiseMergeDBs merges two databases of the same type of
        element into a paired database.  The precondition states
        that the two input DBs must have the same number of
        elements.  The postcondition states that the resulting
        output, dp, must consist of pairs of all the elements that
        are the two input databases.
   )*
end PairwiseMergeDBs
```

It is important to note that universal quantification *does not deliver elements in any specific order*. Given this, the postcondition for the PairwiseMergeDBs example may be weaker than it appears at first reading. Namely, it specifies that dp contains all possible pairs of elements from d1 and d2, where the paired elements were selected from the databases in no guaranteed order. Suppose, for example, that d1 = [r1, r2, r3], and d2 = [r4, r5, r6]. Then the postcondition specifies that the output dp = [[r1,r4], [r1,5], [r1,r6], [r2,r4], [r2,r5], [r2,r6], [r3,r4], [r3,r5], [r3,r6]], where the order of the pairs in dp is not specified.

Suppose that only ordered pairs were desired, such that each pair contains the *ith* element from each of the inputs. This could be specified in a number of ways, including as follows:

```
operation PairwiseMergeDBs
    ...
    postcondition:
        forall (e1,e2: Elem)
            if (e1 in d1) and (e2 in d2)
            then
                exists (ep: ElemPair) ((ep in dp) and
```

```
        PositionOf(ep.e1, d1) = PositionOf(ep.e2, d2))
    ...
end PairwiseMergeDBs
```

where *PositionOf* is the following auxiliary function that outputs the ordinal position of an Elem within a DB:

```
function PositionOf(e:Elem, d:DB):(n:number) =
    if d = nil
    then 0
    else if e = d[1]
         then 1
         else PositionOf(e, d[2:#d]) + 1;
```

8.4.2. Combining Predicative and Equational Specification

The following example shows how a sorted, keyed database can be defined using a combination of equational and predicative specification:

```
object Elem is
    components: k: Key, ElemValue;
    description: (*
        A DB element with an internal key.
    *)
end Elem;

object Key is string;

object SKDB is
    components: Elem*;
    operations:
        Insert: (SKDB, Elem) -> (SKDB),
        Delete: (SKDB, Key) -> (SKDB),
        Find: (SKDB, Key) -> (Elem),
        EmptySKDB: () -> (SKDB),
        FindNth: (SKDB, number) -> (Elem),
        SortDB: (SKDB, Key) -> (SKDB);
    equations:
        var d: SKDB; e:Elem; i:number;
        Find(EmptyDB, k) == EmptyElem();
        Find(Insert(d, e), k') ==
            if e.k=k' then e else find(d, k');
        Delete(EmptyDB, k) == EmptyDB;
        Delete(Insert(d, k), k') ==
            if k=k'
            then Delete(d, k')
            else Insert(Delete(d, k'), k, e);
        FindNth(Insert(d,e), i) ==
          if i = 1 then e
          else FindNth(d,i-1);
end SKDB;

operation SortDB is
    inputs:    d:SKDB, k:Key;
    outputs:   d':SKDB;
    precond:   ;
    postcond:  forall (i,j:number)
             if i<j then FindNth(d',i).k < FindNth(d',j).k;
```

```
        description: (* Sort a database *)

    end SortDB;

    operation FindNth is
        inputs:d:SKDB, n:number;
        outputs: e:Elem;
        description: (* Find the DB element at position n. *)
    end FindNth;
```

Of particular note in this example is the universal quantifier in the SortDB postcondition. An English paraphrase of this postcondition is as follows: "For all pairs of numbers i and j, if i is less than j then the key at position i is less than the key at position j, where the key at position n is delivered by the FindNth operation."

8.4.3. More on Auxiliary Functions

A negative critique of keyed database specification is that FindNth might best be left invisible to end-users. That is, there may be no reason that users need to find the nth element in a DB. Rather, users simply need the DB sorted and to be able to find elements by key. Even if this is the case, we still need FindNth in order to fully specify the definition of SortDB.

It was noted earlier that specification is a battle between saying too little and saying too much. In the case of FindNth, the battle has been lost if FindNth is not a necessary user-level operation, since FindNth is still a necessary specification-level function. In the example above, FindNth was specified equationally as a visible operation. The alternate definition as an auxiliary function is the following:

```
function FindNth(d:SKDB, n:number):(e:Elem) = d[n];
```

In some cases, it may turn out that what at first appears to be an auxiliary function is in fact a legitimate user-level operation. When this happens, the process of formalizing a specification has helped uncover an incompleteness originally present before the formal definition was considered. In other cases, auxiliary functions should best be left invisible to the user, in which case they remain genuinely auxiliary. In all cases, specifiers must consider carefully when auxiliary functions are necessary.

A noteworthy property of auxiliary functions is that their definitions are *constructive*. That is, the body of the function constructs an actual value that the function outputs. Such constructive definitions are in contrast to operations that are defined solely in terms of pre- and post-conditions. A postcondition does not construct a value, but rather states a property that an *assumed* constructed value must meet.

The astute reader will notice that with the introduction of constructive functions, RSL has the expressive power of a functional programming language, such as LISP or ML. Hence, using constructive functions, it is possible for specifications to become very much like programs. This is clearly not the intent. Specifiers should always be mindful that the fundamental goal of specification is to define *what* a system does, not how it works. Auxiliary functions should be used judiciously, so that RSL specifications remain free of unnecessary program-level detail.

9. Execution

The RSL language defined in the preceding sections is not executable. Operations are specified strictly in terms of conditions that must be true upon entry and exit. Objects are specified by equational relationships between operations, not by operations that manipulate data structures.

In writing a specification, it is often convenient to provide prototype implementations for some of the specified operations. This allows experimentation with implementation alternatives, and testing of concrete ideas.

There are a number of possibilities for rendering a specification executable. One successful technique is employed in the OBJ3 language [Goguen 88]. The OBJ3 approach to execution is to interpret an equational specification using term rewriting. The concept of term rewriting was introduced in Section 8.3 above. Execution in OBJ3 uses the same basic rewriting technique as described there. The advantage of equational interpretation is that the specification can be executed directly, with no additional definitions required specifically for execution. A significant disadvantage of equational execution is inefficiency, since term rewriting is quite slow when implemented via software interpretation. Another disadvantage is that only the equational form of the specification is executable, since predicative specifications cannot be interpreted using the same form of rewriting that is used for equations.

Another approach to executable specification is employed in the FASE specification language. In FASE, an interpreter for predicative specification has been developed. The FASE approach to execution has the same advantage as the OBJ3 approach. Viz., the FASE specification is directly executable. A disadvantage of the FASE system is that not all forms of quantification can be executed, most particularly existential. In addition, some forms of universal quantification lead to quite inefficient execution.

A simpler approach to execution is taken for RSL. Rather than employing sophisticated interpretation of equational or predicative specification, RSL allows the specifier to use a subset of the RSL expression language to specify executable bodies for operations. The executable subset of RSL is comparable to functional programming languages, such as pure Lisp and ML. In terms of expressive power, executable RSL is closer to Lisp than to ML, since the RSL interpreter does not perform type inference.

9.1. Implementation Modules

To provide an executable body for one or more operations, the RSL specifier adds an *implementation* module to a specification. Each implementation module must correspond by name to an existing specification module. Consider the following example:

```
implmentation module PersonDatabase;

    operation AddRecord (pdb: PersonDatabase, pr: PersonRecord)
                                    : pdb': PersonDatabase;
        let pdb' = pdb + pr;
    end AddRecord;

    operation DeleteRecord (pdb: PersonDatabase, pr: PersonRecord)
                                    : pdb': PersonDatabase;
        let pdb' = pdb - pr;
    end DeleteRecord;

    operation FindRecord (pdb: PersonDatabase, pr: PersonRecord, n: Name)
                                    : pr: PersonRecord;
        forall (pr1 in pdb) (
            if (pr1.Name = n) then let pr = pr1;
        )
    end FindRecord;
```

```
    operation CreateDatabase() : pdb: PersonDatabase;
        let pdb = [];
    end CreateDatabase;

end PersonDatabase;
```

This is a companion implementation for the PersonDatabase example presented earlier in Sections 3.1 and 3.2. The body of each operation defines how to compute a value for the operation outputs. Syntactic and semantic details of executable expressions follow.

9.2. Executable Expressions

The foundation of executable expressions is identical to the expressions defined in Sections 7.1 through 7.4 of the manual. These sections define the following:

- variables
- functional, arithmetic, and boolean expressions
- list operations
- composite object selector operations

Added to this foundation are three additional forms of executable expression:

- the **let** expression
- executable **forall**
- expression sequencing

The **let** expression has the following format:

let *object-selector = executable expression*

where *object-selector* is a variable or an object selector expression as defined in Section 7.4. The **let** expression binds a value to a variable or to the selected component of a variable.

Executable **forall** has the following format:

forall *(x in list) expression-sequence*

The meaning of executable forall is as follows. The *expression-sequence* will be executed zero or more times, based on the number of elements of in *list*. Before the ith execution, the expression "**let** *x = list[i]* is executed. That is, the value of *x* is set successively to each element in *list*, and the *expression-sequence* is successively executed. Note that there is a significant semantic difference between executable **forall** versus the logical **forall** used in a predicate. As noted in Section 8.4.1, the logical **forall** does not deliver elements in any guaranteed order. In contrast, executable **forall** does deliver elements in order, such that the ith execution works on the ith element of the list.

An expression sequence is simply a sequence of executable expressions separated by semicolons. The sequence is executed in order.

Executable object values are defined using the syntax of concrete objects, defined in Section 3.7. For example, the following is a typical executable expression sequence:

```
let s = [1,2,3];
let s[1] = 10;
let s[2:3] = [20, 30, 40, 50];
```

For this to be a legal sequence, the type of variable s must be number*. The result of this sequence is a value of [10, 20, 30, 40, 50] for s.

Executable expressions are put to use within the bodies of executable operations. The definition of an executable operation has the following general form, similar to an auxiliary function:

> **operation** *name* (*list of inputs*) : (*list of outputs*);
>   **var** *var_name:object_name, ...* ;
>   *executable-expression-sequence* ;
> **end** *name*

The variable declarations within the operation define local variables that may be used within the executable expressions that follow the declarations. The body of the function must contain at least one let expression for each of the operation outputs. These let expressions define the output values that are computed by the operation.

10. Concluding Remarks

This manual has described a general-purpose requirements specification language. The language is suitable for specifying requirements and external functionality of a computer-based system. The language can be used for the software and/or hardware components of a system.

Development of the language is ongoing. Enhancements scheduled for future release include the following:

- Fully formal definition of RSL via mapping to an existing formal specification language, such as EHDM [Rushby 91] or HOL [Gordon 85].
- Execution via term-rewriting of equational specifications, as provided in OBJ [Goguen 88].
- Support for the formal specification of graphical user interfaces, as described in [Fisher 91].

References

[Fisher 91] G. L. Fisher and D. A. Frincke, "Formal Specification and Verification of Graphical User Interfaces," *Proceedings of the Hawaii International Conference on System Science*, January 1991.

[Fisher 93] G. L. Fisher and G. C Cohen, "Tools Reference Manual for a Requirements Specification Language (RSL), Version 2.0", NASA Contractor Report 191461, September 1993

[Frincke 92] D. A. Frincke; D. A. Wolber, G. L. Fisher, and G. C. Cohen, "Requirements Specification Language (RSL) and Supporting Tools", NASA Contractor Report 189700, July 1992.

[Goguen 88] J. A. Goguen and T. N. Winkler, "Introducing OBJ3", SRI International Technical Report, Palo Alto, CA, August 1988.

[Greenspan 82] S.J. Greenspan, J. Mylopoulos, and A Borgida, "Capturing More World Knowledge in the Requirements Specification", *Proceedings of the Sixth International Conference on Software Engineering*, 1982.

[Gordon 85] M. Gordon, "A Proof Generating System for Higher-Order Logic", University of Cambridge Computer Laboratory, January 1987.

[Guttag 85] J. Guttag, J. J. Horning and J. M. Wing, "The Larch Family of Specification Languages", *IEEE Software*, May 1985.

[Ross 77] D. T. Ross, "Structured Analysis (SA): A Language for Communicating Ideas", *IEEE Transactions on Software Engineering*, January 1977.

[Rushby 91] J. Rushby, "The EHDM Reference Manual", SRI International Technical Report, Palo Alto, CA, 1991.

[Stroustrup 91] B. Stroustrup, *The C++ Programming Language, Second Edition*, Addison-Wesley, 1991.

[Teichroew 77] D. Teichroew and E. A. Hershey III, "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems", *IEEE Transactions on Software Engineering*, January 1977.

[Wirth 85] N. Wirth, *Programming in Modula-2, Third Edition*, Springer-Verlag, 1985.

Appendix A: Summary of Entity Definition Forms

The examples in the body of the manual show how objects and operations can be defined in a number of forms. This Appendix summarizes each of the definitional forms.

A.1 Object Definition Forms

The most complete form of object definition is the following:

object *name* is
  [**components:** ...;]
  [**operations:** ...;]
  [**equations:** ...;]
  [**description:** ...;]
  [<user-defined attributes>; ... ;]
end <name>

where the square brackets denote optional terms. This long-form specifies a composite type if the **components** attribute is non-empty, or an opaque type if the **components** attribute is missing or defined as **empty**.

The following is a shorter definition form used to specify atomic types:

object *name* is *name* [
  [**operations:** ...;]
  [**equations:** ...;]
  [**description:** ...;]
  [<user-defined attributes>; ... ;]
end <name>]

Notice in this shorter form that the components attribute is missing. The other attributes remain optional in the short form.

Replacing the keyword **is** with '=' defines a concrete value rather than an abstract type. The general format for a concrete object definition is as follows:

object *name = expression* [
  [**description:** ...;]
  [<user-defined attributes>; ... ;]
end <name>]

Notice that components, operations, and equations attributes are all missing in a concrete value definition. Formally, it is inappropriate to include these attributes for a concrete value, so their inclusion is disallowed syntactically.

The simplest form of object specification is a fully opaque type, defined as:

object *name ;*

A.2 Operation and Function Definition Forms

The general forms for operation definition are analogous to the object forms. The long-form operation definition is the following:

operation *name* is
  [**components:** ...;]
  [**inputs:** ...;]

```
    [outputs: ...;]
    [preconditions: ...;]
    [postconditions: ...;]
    [description: ...;]
    [<user-defined attributes>; ... ;]
end <name>
```

This form defines a user-visible operation with optional attributes. A non-user-visible auxiliary function can be defined in two forms, the first of which is:

```
function name is
    [inputs: ...;]
    [outputs: ...;]
    [preconditions: ...;]
    [postconditions: ...;]
    [description: ...;]
    [<user-defined attributes>; ... ;]
end <name>
```

This form defines a non-constructive auxiliary function. The formal specification of such functions is given with pre/postconditions or equationally.

The final function definition form is:

```
function name = (inputs):(outputs) = epxr
    [preconditions: ...;]
    [postconditions: ...;]
    [description: ...;]
    [<user-defined attributes>; ... ;]
end <name>
```

This form defines a constructive auxiliary function, where the given expression defines the concrete value computed by the function. Notice that a constructive function may optionally include pre/postconditions so that a both constructive and predicative definition can be supplied if desired. The two definitions can provide complementary but equivalent specifications.

Appendix B: Keyword Synonyms

To provide flexibility in the textual style of an RSL specification, most keywords have abbreviated forms that are synonymous with the longer forms of the keyword. The following table summarizes all keyword synonyms:

Full Keyword	Synonymous Abbreviations
and	','
axiom	ax
collection	list, list of, '*' (as a postfix operator)
components	parts
exists	exist, there exist, there exists
function	func
iff	'<=>'
implies	'=>'
inputs	in
object	obj
operation	op
operations	ops
outputs	out
postconditions	postcond, post
preconditions	precon, pre
variable	var

Appendix C: Functional Definition of Relational Attributes

Formally, relational attributes can be viewed as "syntactic sugaring" for a functional definition of formal relations. RSL users who prefer to limit the number of constructs used in a specification may prefer not to use relational attributes.

In general, the relation between two objects is represented as a function (or operation) with an appropriate signature. Consider the following bi-directional relation:

```
define object attribute R;
object O1 is
    ...
    R: O2;
end O1;
object O2 is
    ...
    R: O1
end O2
```

The functional representation is as follows:

```
object O1 is
    ...
end O1;
object O2 is
    ...
end O2;
function O1_R_O2(O1):(O2);
function O2_R_O1(O2):(O1);
```

Multi-valued relations, such as

```
object O3 is
    ...
    R: O1,O2;
end O3;
```

are represented as multi-valued functions or multiple, suitably-named, single-valued functions. The functional representation is clearly bulkier than the attribute-based representation, since in the former, one function needs to be defined for each separate reference to a single relation.

An additional consideration regarding relational definitions concerns the future enhancement of the RSL development environment. A near-term goal is to provide a formal verification environment for RSL, so that properties of a specification can be mechanically verified. The most effective means to provide such verification is to translate RSL into an existing formal language for which mechanized verification support already exists. Candidates for the target of translation include EHDM [Rushby 91] and HOL [Gordon 87]). These and similar systems do not provide direct verification support for relational definitions. Therefore, translation of relations into functional form will be necessary if mechanized verification support is to be provided for RSL in the foreseeable future.

Appendix D: Complete RSL Syntax

```
top_level_def:
        spec_unit
                |
        entity_spec
                |
        attr_def

spec_unit:
        module_heading entity_spec_list ';' 'end' spec_name_ender ';' |
        module_heading attr_defs ';' entity_spec_list ';' 'end' spec_name_ender
  ';' |
        module_heading 'end' spec_name_ender ';' |
        module_heading attr_defs ';' 'end' spec_name_ender ';' |
        entity_spec ';'

module_heading:
        'module' spec_name ';' |
        'module' spec_name ';' imports |
        'module' spec_name ';' exports |
        'module' spec_name ';' imports exports

imports:
        import |
        import imports

import:
        'from' name 'import' ident_list ';' |
        'import' ident_list ';'

exports:
        export |
        export exports

export:
        'export' 'qualified' ident_list ';' |
        'export' ident_list ';'

entity_spec_list:
        entity_spec |
        entity_spec_list ';' entity_spec

entity_spec:
        object_spec |
        operation_spec |
        formal_decl
```

```
object_spec:
        obj_heading 'is' obj_body 'end' obj_name_ender |
        obj_heading instance 'is' obj_body 'end' obj_name_ender |
        obj_symbol name_type_pair '=' obj_expr |
        obj_symbol name_type_pair '=' obj_expr obj_attributes 'end' obj_name_ender |
        obj_heading 'is' obj_name |
        obj_symbol obj_name

obj_heading:
        obj_symbol obj_name |
        obj_symbol class obj_name

operation_spec:
        op_heading 'is' op_body 'end' op_name_ender |
        op_heading instance 'is' op_body 'end' op_name_ender |
        op_symbol op_name '(' name_type_list ')' ':' '(' name_type_list ')' '='
          expr |
        op_symbol op_name '(' name_type_list ')' ':' '(' name_type_list ')' '='
          expr precond postcond op_attributes 'end' op_name_ender

op_heading:
        op_symbol op_name |
        op_symbol class op_name

class:
        'class'

instance:
        'instance' 'of' class_name_list

obj_body:
        parts ops eqns obj_attributes |
        name ops eqns obj_attributes |
        parts ops eqns |
        name ops eqns

op_body:
        parts inputs outputs precond postcond op_attributes

parts:
        /* empty */ |
        parts_spec or_op parts_spec |
        prefix_list_op parts_spec |
        parts_spec postfix_list_op |
        name_type_pair |
        name ':' '(' parts_spec ')' |
        '(' parts_spec ')'
```

```
and_op:
        'and' |
        op_name ':' op_parms

op_parms:
        '(' sig_args_list ')' '->' '(' sig_args_list ')'

name_type_list:
        /* empty */ |
        prefix_list_op ins_parts_spec |
        ins_parts_spec postfix_list_op |
        init_name_type_pair

obj_attributes:
        obj_attribute |
        obj_attributes ';' obj_attribute

obj_attribute:
        attr_name ':' 'text'

attr_name:
        name

op_attributes:
        /* empty */ |
        obj_attributes

name_type_pair:
        name |
        name ':' obj_name

name_obj_pair:
        name |
        name ':' obj_name

init_name_type_pair:
        name_type_pair |
        name_type_pair ':=' obj_expr

spec_name:
        name

spec_name_ender:
        /* empty */ |
        name

class_name_list:
```

```
   class_name |
        class_name_list ',' class_name

class_name:
        name

obj_name:
        name

obj_name_ender:
        /* empty */ |
        name

op_name:
        name

op_name_ender:
        /* empty */ |
        name

formal_decl:
        var_symbol var_decls ';' |
        function_decl ';' |
        ax_symbol ax_decls ';'

var_decls:
        var_decl |
        var_decls ';' var_decl

var_decl:
        var_name_list ':' obj_name

function_decl:
        function_heading '(' name_type_list ')' ':' '(' name_type_list ')'
                                        '=' expr |
        function_heading '(' name_type_list ')' ':' '(' name_type_list ')'
                '=' expr precond postcond op_attributes 'end' op_name_ender |
        function_heading '(' name_type_list ')' ':' '(' name_type_list ')' |
        function_heading '(' name_type_list ')' ':' '(' name_type_list ')'
                'is' precond postcond op_attributes 'end' op_name_ender

function_heading:
        'function' op_name |
        'func' op_name

precond:
        /* empty */ |
```

```
    pre_symbol ':' ';' |
        pre_symbol ':' expr ';' |
        pre_symbol error ';'

postcond:
        /* empty */ |
        post_symbol ':' ';' |
        post_symbol ':' expr ';' |
        post_symbol error ';'

expr:
        expr 'and' expr |
        expr 'or' expr |
        expr 'implies' expr |
        expr 'iff' expr |
        'not' expr |
        'if' expr 'then' expr |
        'if' expr 'then' expr 'else' expr |
        'forall' '(' name_obj_list ')' expr |
        exists_symbol '(' name_obj_list ')' expr |
        rel_expr

rel_expr:
        expr rel_bin_op expr  %prec '=' |
        arith_expr

arith_expr:
        arith_expr arith_add_op arith_expr |
        arith_expr arith_mult_op arith_expr |
        arith_expr arith_exp_op arith_expr |
        arith_pre_op arith_expr |
        selector_expr |
        index_expr |
        functional_expr |
        obj_expr |
        '(' expr ')'

selector_expr:
        arith_expr select_op arith_expr          %prec '.'

index_expr:
        arith_expr '[' expr ']' %prec '[' |
        arith_expr '[' expr ':' expr ']'

functional_expr:
        op_name '(' op_args ')'
```

```
op_args:
        /* empty */ |
        op_arg |
        op_arg ',' op_args

sig_args_list:
        /* emtpy */ |
        sig_args

sig_args:
        sig_arg |
        sig_arg ',' sig_args

op_arg:
        expr

sig_arg:
        name_type_pair

rel_bin_op:
        '=' |
        '<' |
        '>' |
        '<>' |
        '<=' |
        '>=' |
        'in'

arith_add_op:
        '+' |
        '-'

arith_mult_op:
        '*' |
        '/' |
        'div' |
        'mod'

arith_exp_op:
        '^'

arith_pre_op:
        '+' |
        '-' |
        '#'

select_op:
```

```
    '.' |
        '?'

eqns:
        /* empty */ |
        eqns_symbol ':' eqn_decls ';' |
        eqns_symbol ':' {EnterEqns();}
                        var_symbol var_decls ';' eqn_decls ';' |
        eqns_symbol error ';'

eqn_decls:
        eqn_decl |
        eqn_decls ';' eqn_decl

eqn_decl:
        lhs '==' rhs ;

lhs:
        functional_expr
                ;

rhs:
        expr
                ;

ax_decls:
        ax_decl |
        ax_decls ';' ax_decl

ax_decl:
        expr

var_name_list:
        var_name |
        var_name_list ',' var_name

var_name:
        name

obj_expr:
        obj_atom
                |
        '[' ']' |
        '[' obj_expr_list ']'

obj_expr_list:
        obj_expr |
```

```
    obj_expr_list ',' obj_expr

obj_atom:
        "double-quoted string of printable ASCII characters" |
        "zero or more digits" |
        "zero or more digits with optional fraction and exponent" |
        'nil' |
        name

name:
        "letter followed by zero or more letters or digits"

attr_defs
        'define' op_symbol 'attribute' ident_list |
        'define' 'attribute' ident_list ;

ident_list:
        var_name_list

obj_symbol:
        'object' |
        'obj'

op_symbol:
        'operation' |
        'op'

parts_symbol:
        'components' |
        'parts'

ops_symbol:
        'operations' |
        'ops'

in_symbol:
        'inputs' |
        'in'

out_symbol:
        'outputs' |
        'out'

var_symbol:
        'variable' |
        'var'
```

```
eqns_symbol:
        'equations' |
        'eqns'

ax_symbol:
        'axiom' |
        'ax'

pre_symbol:
        'preconditions' |
        'precondition' |
        'pre'

post_symbol:
        'postconditions' |
        'postcondition' |
        'post'

exists_symbol:
        'there' 'exists' |
        'there' 'exist' |
        'exists' |
        'exist'
```

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188

1. AGENCY USE ONLY (Leave blank)

2. REPORT DATE
November, 1993

3. REPORT TYPE AND DATES COVERED
Contractor Report

4. TITLE AND SUBTITLE
Tools Reference Manual for a Requirements Specification Language (RSL), Version 2.0

5. FUNDING NUMBERS
C NAS1-18586
WU 505-64-10-07

6. AUTHOR(S)
Gene L. Fisher*
Gerald C. Cohen

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Boeing Defense and Space Group
P.O. Box 3707, M/S 4C-70
Seattle, WA 98124-2207

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23681-0001

10. SPONSORING MONITORING AGENCY REPORT NUMBER
NASA CR - 191461

11. SUPPLEMENTARY NOTES
Langley Technical Monitor: Sally C. Johnson
Task 11 Report
*California Polytechnic State University, San Luis Obispo, CA

12a. DISTRIBUTION AVAILABILITY STATEMENT
Unclassified - Unlimited

Subject Category 60

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)
This report describes a general-purpose Requirements Specification Language, RSL. The purpose of RSL is to specify precisely the external structure of a mechanized system and to define requirements that the system must meet. A system can be comprised of a mixture of hardware, software, and human processing elements. RSL is a hybrid of features found in several popular requirements specification languages, such as SADT (Structured Analysis and Design Technique [Ross 77]), PSL (Problem Statement Language [Teichroew 77], and RMF (Requirements Modeling Framework [Greenspan 82]). While languages such as these have useful features for structuring a specification, they generally lack formality. To overcome the deficiencies of informal requirements languages, RSL has constructs for formal mathematical specification. These constructs are similar to those found in formal specification languages such as EHDM (Enhanced Hierarchical Development Methodology [Rushby 91]), Larch [Guttag 85], and OBJ3 [Goguen 88].

14. SUBJECT TERMS
Requirements Specification Language
Formal Specification
Classes

15. NUMBER OF PAGES
70

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT
Unclassified

18. SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

19. SECURITY CLASSIFICATION OF ABSTRACT

20. LIMITATION OF ABSTRACT